IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

PROMOS TECHNOLOGIES, INC.,	)	
	)	
Plaintiff,	)	
	)	
v.	)	C.A. No. 06-788 (JJF)
	)	
FREESCALE SEMICONDUCTOR, INC.,	)	
	)	
Defendant.	)	
	)	

APPENDIX TO FREESCALE'S OPENING CLAIM CONSTRUCTION BRIEF VOLUME II; EXHIBITS RELATING TO THE CHAN PATENTS MOSEL PRODUCT SPECIFICATIONS

MORRIS, NICHOLS, ARSHT & TUNNELL LLP
Mary B. Graham (#2256)
James W. Parrett, Jr. (#4292)
1201 N. Market Street
P.O. Box 1347
Wilmington, DE 19899-1347
302.658.9200

Attorneys for Freescale Semiconductor, Inc.

OF COUNSEL:

David L. Witcoff
Kevin P. Ferguson
John M. Michalik
JONES DAY
77 West Wacker
Chicago, IL 60601-1692
312.782.3939

F. Drexel Feeling
JONES DAY
North Point
901 Lakeside Avenue
Cleveland, OH 44114-1190
216.586.3939

Dated: November 6, 2007

TABLE OF EXHIBITS

EXHIBIT	DESCRIPTION
1	MS441 & MS443 Data Sheet
2	MS443 Specification Sheet
3	MS486 Specification Sheet
4	Mosel's Burstram Architecture Specification
5	Appendix Part 1 Mosel's abandoned patent

1307661

EXHIBIT 1

MOSEL

MS441 Cache Controller

PRELIMINARY

SimulCache chipset

FEATURES

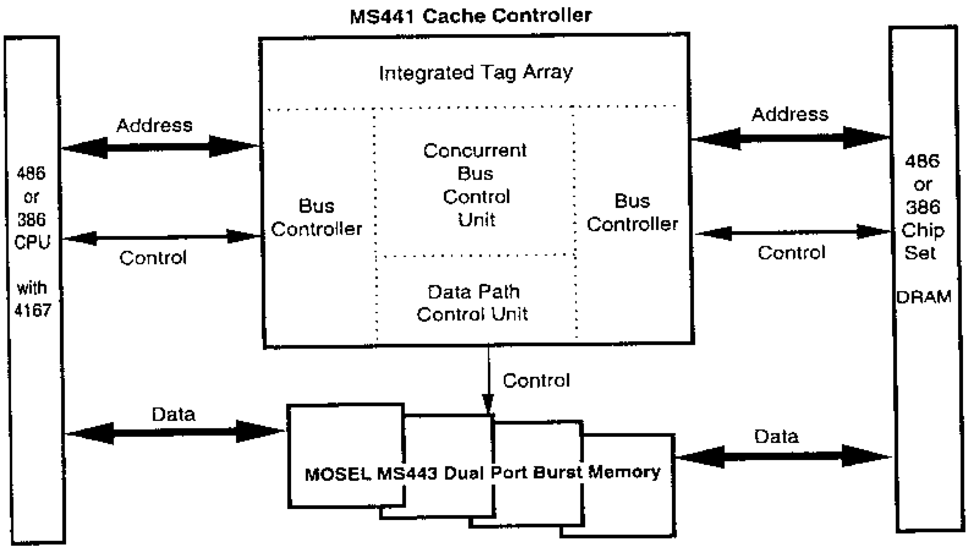
- High Performance Cache Controller optimized for 486 Secondary cache or 386 Primary cache applications
- Integrates two 386/486 bus controllers in combination with Dual Port Burst Memories for Concurrent Write Back Architecture
- Concurrency permits background write back while read and write cache hits continue uninterrupted
- Provides 96% read hit rate and 99.8% write hit rate (Converts a write miss into a write hit in one cycle)
- Compatible with 386 or 486 chip set products. Burst read provides fetch ahead for 386 applications
- Introduces an optional superset of the 486 System bus supporting burst writes as well as burst reads
- Two way set associative with 2K tag entries
- Four programmable cache regions, each with non-cachable, cachable, and cachable write protect mode
- Control signal provided to support local bus peripherals
- Supports Weitek 4167 coprocessor
- Primary and secondary cache coherency supported with concurrent bus snooping on DMA cycles
- 184 pin PQFP - Sub Micron CMOS technology

DESCRIPTION

The MOSEL MS441 is a high-performance, high integration cache controller for use with the 80486 or 80386 in high performance computer systems. With its Concurrent Write Back Architecture, it allows unlimited simultaneous CPU and System Bus accesses resulting in an exceptionally high write hit rate of 99.8%, and a read hit rate of 96%. This has a high value in 386 systems but particularly 486 systems which have more writes than reads on the system bus.

The Concurrent nature of the Cache Subsystem is facilitated by MOSEL's MS443 Dual Port Burst Memories. The Dual Concurrent Bus Controllers in the MS441 manage the high speed data path from CPU to Cache Memory to the System.

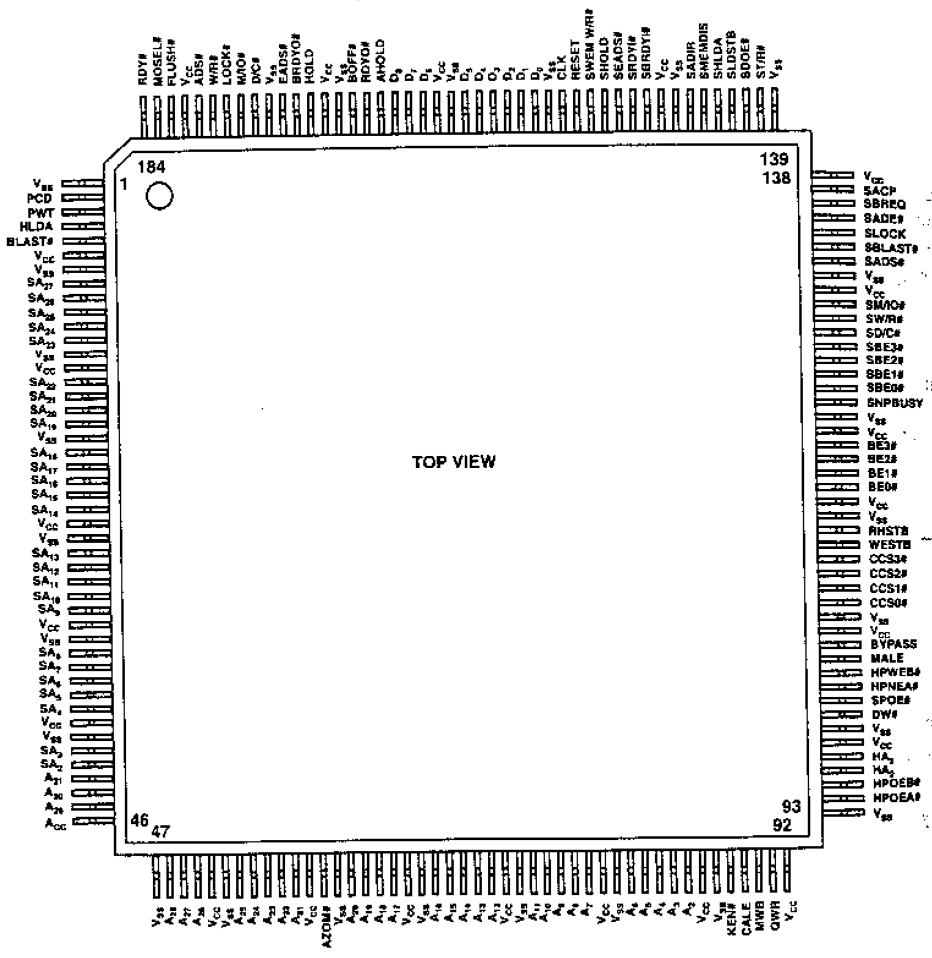
In order to improve data throughput, the MS441 introduces the Burst Write to the system bus. This is an optional mode, which if utilized reduces bus saturation on the system side.



9

PRS DEL 015762

MS441 Pinout (Top View)



MOSEL

MS443

PRELIMINARY

Intelligent Dual Port Burst Memory

FEATURES

- High Performance Intelligent Memory optimized for fast burst read and burst write operations
- Compatible with the MOSEL MS441 Concurrent Write Back Cache Controller
- Supports 386 and 486 microprocessors to 25 Mhz and 33 Mhz at Zero Wait States
- Intelligent Dual Port Burst Architecture achieves a max bandwidth of 256 MBytes /sec (8x higher than SRAM)
- On chip address and data registers optimize the data path for concurrent dual bus accesses
- Exceptionally fast register access time – 8ns
- 144Kbit Memory Array is arranged in two sets of four 2Kx9 blocks, supporting 2 way set associativity and simultaneous latching of 16 Bytes in one CPU clock
- Data path registers channel data into, out of, and around the memory array in single clock operations.
- Bypass Mode allows fast data streaming from System Port to CPU Port on read miss operations (5ns)
- 64KByte configuration achieved with 4 Devices
- Internal pipeline makes self-timed write possible
- 64 pin PQFP or PLCC - Sub micron CMOS technology

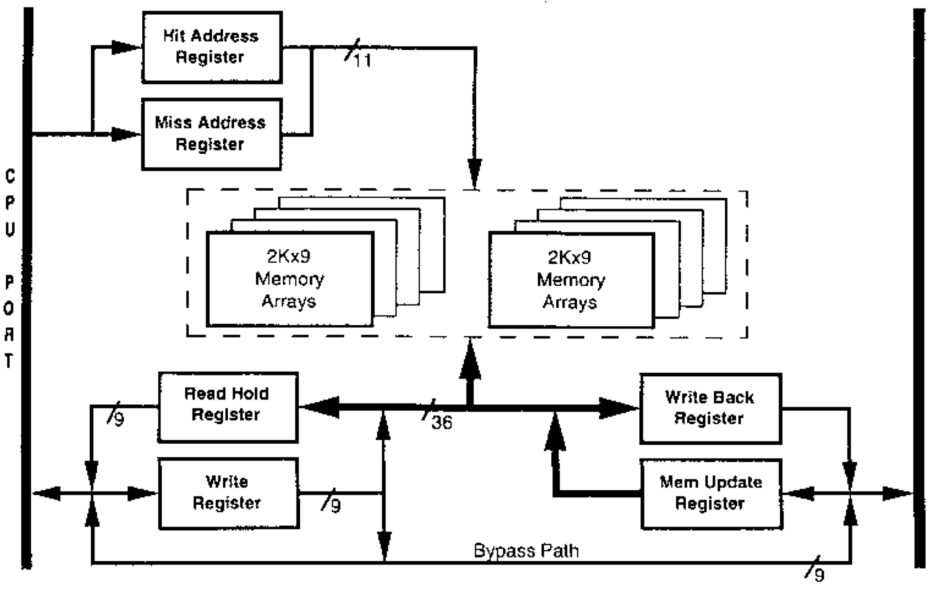
DESCRIPTION

The MOSEL MS443 is an Intelligent Data Path and Cache Memory device optimized for use with MOSEL's MS441 Cache Controller. Dual Ports combined with an intelligent internal data path create the enabling technology for Concurrent Write Back Cache Architecture.

The device contains a sophisticated combination of 36 bit read and write data registers for each of its dual ports, plus hit and miss address registers.

The advanced data path management techniques used inside the MS443 make possible a write hit rate of 99.8%, and a read hit rate of 96%. The extraordinarily high write hit rate is of particular value in 486 systems which have bus activity comprised of75% writes.

This cache memory subsystem has an internal data path similar to that of the 486 CPU. With 4 devices, the internal data bus is 128bits wide and is capable of single clock transfers. This equates to a 533 MByte per second maximum internal transfer rate. The device adds dual 36 bit buses externally which are suited to 486 style burst transfers or 386 style transfers.



MOSEL Corporation 914 West Maude Avenue, Sunnyvale, CA 94086 U.S.A. 408-733-4556

MS443 Pinout (Top View)

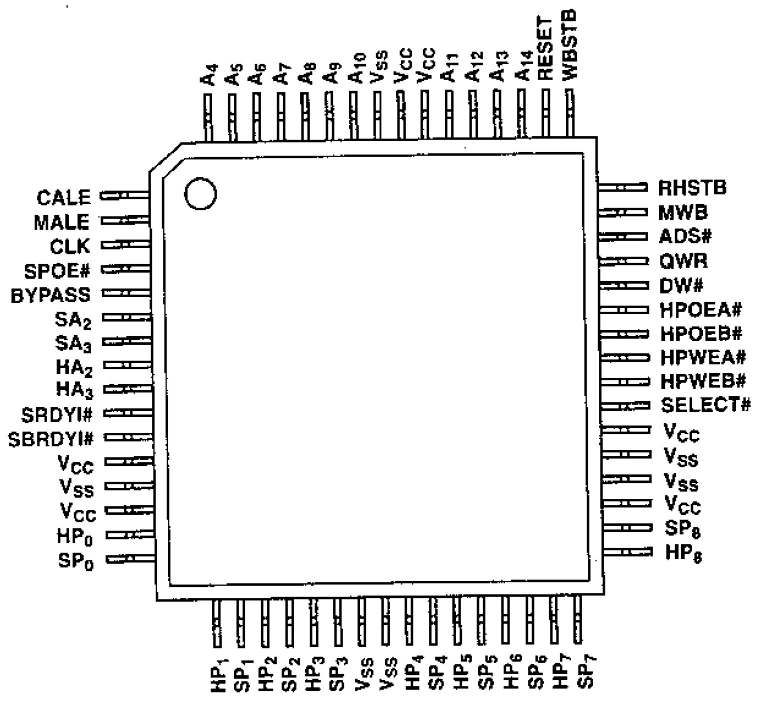


EXHIBIT 2

MOSEL **MS82C443**

ADVANCE INFORMATION
JANUARY 1991

SimulCache™ Burst-RAM

FEATURES

- Complete 0-wait-state 80486 SimulCache™ Solution
  - 64K Byte SimulCache
    - 1 MS82C441
    - 4 MS82C443
  - 128K Byte SimulCache
  - 256K Byte SimulCache
- High Performance
  - Supports true 80486 burst reads and 80486 writes at 25 and 33 MHz, with future migration to 40 and 50 MHz
  - Dual-access SimulCache supports simultaneous processor/cache and cache/main memory read/write cycles
    - Cache hits and memory fetches run in parallel to boost hit rate and reduce miss penalty
  - Supports burst read and burst write between cache and main memory
    - Also supports non-burst quad word read/write cycles
    - "Demand word first" priority scheme minimizes cache miss penalty
  - Supports write-back cache update strategy with hidden write back cycles
    - 0-wait-state write hits and misses
    - Reduced system bus traffic
    - Word qualify signal prevents "dirty" word overwrite by stale data
- High Integration
  - On-chip hit and miss address registers
  - 2 x 2K x 36 internal organization
    - Two banks of 2K x 36 divided into four subarrays of 2K x 9
    - Four device cache supports 128-bit line burst
- Flexibility
  - Supports 486 or sequential order for main memory read/write
    - Permits resequencing of 80486 memory fetches to eliminate the need for interleaved memory
  - Compatible with burst, bank interleaved, or sequential non-burst main memory
  - Unique Dual Ported Architecture decouples host and system data buses
    - Main memory/system bus can be operated at different speed than processor
    - Easy upgrade of processor module without affecting main memory design
  - Self-timed feature supports 80486 late write timing
- 80486 Special Features
  - Synchronous 80486 interface
  - Converts to 486 burst order on host port
  - Supports PWT and PCD cycles
  - Dual nine bit data ports support 486 parity functions

DESCRIPTION

The MOSEL MS82C443 SimulCache Burst-RAM is a high-performance CMOS static RAM with a proprietary dual-access architecture optimized for use as a cache subsystem data buffer in 80486 systems. The MS82C443 supports burst mode read operations between cache and the host and burst or non-burst reads amd writes between cache and main memory to allow the 80486 to run at its full potential. The dual-access architecture allows cache hits to proceed in parallel with main memory accesses, improving hit rate and processor performance. The architecture also permits write-back line replacement cycles to be hidden from the processor, providing a significant performance improvement over alternative write-back implementations.

[illegible]

Table of Contents

Section 1. MS82C443 Pinout (Top View)

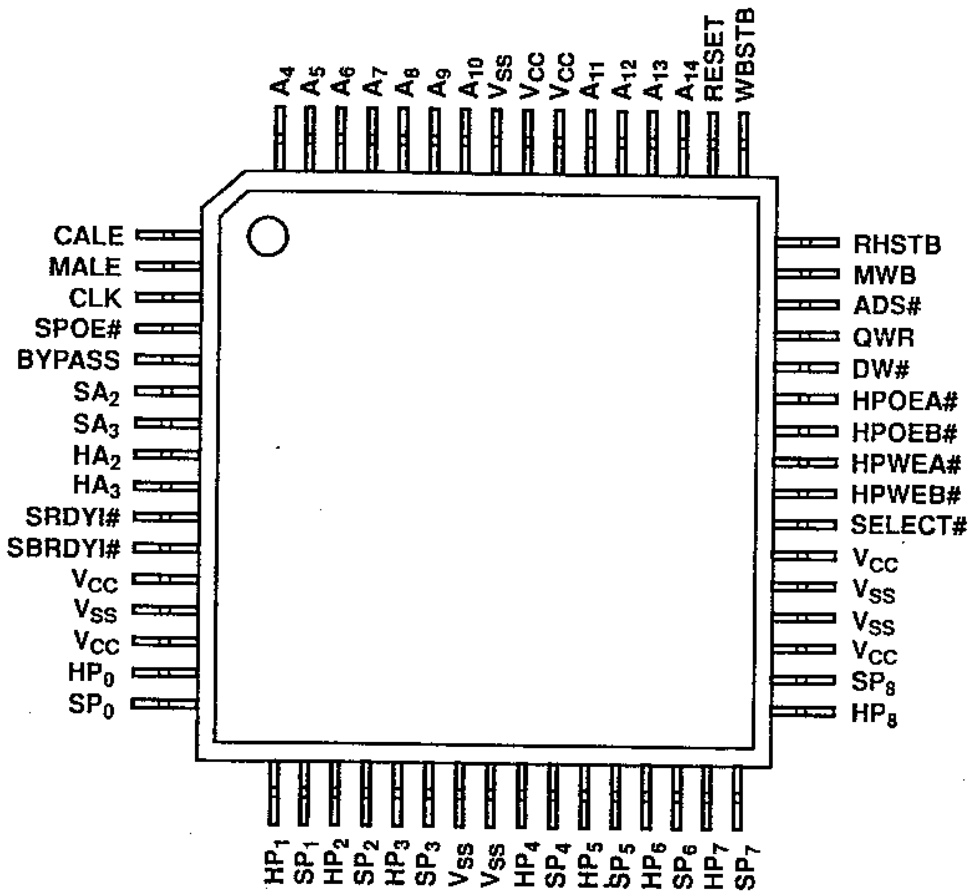


Figure 1

Section 2. MS82C443 Pin Descriptions

Signal	Type	Description
ADDR<14:4>	Inputs	**486 Local Address Bus.** These bits address one of 2048 entries of 32-bit double-words in each burst-RAM data array.
ADS#	Input	**Address/Data Strobe.** This is the latch enable strobe for CPU-generated bus cycles. The falling edge of this signal creates a flow-through mode for the HADDREG. The rising edge of ADS# latches the address into HADDREG.
CALE	Input	**Controller Address Latch Enable.** This is the latch enable strobe for controller-generated bus cycles. CALE opens the HADDREG level latch to ADDR.
MALE	Input	**MS82C443 Miss Address Latch Enable.** This signal should be asserted after ADS# or CALE activation if a miss for either read or write cycles has been detected. MALE performs several functions. First, it latches the address in HADDREG into MADDREG. It also inhibits any host port writes from being performed on clock edges that it is sampled high. In addition, MALE latches bank select information in order to direct a subsequent update into the cache array. Activation of MALE with either HPOEA# or HPWEA# indicates that the next quad write (QWR) operation will update bank A, while MALE and either HPOEB# or HPWEB# selects bank B. Bank selection, as described above, must be qualified with assertion of the SELECT# input. MALE has a higher priority than does the MWB input. Assertion of both MALE and MWB on the same clock edge will result in recognition of MALE, but not MWB.
RESET	Input	**MS82C443 Reset.** Reset should be active for at least four CLK clocks for MS82C443 to complete reset. Reset will clear all mask and valid bits.
WBSTB	Input	**Write Back Strobe.** The rising edge of WBSTB results in the MS82C443 data entry associated with MADDREG being latched into WBREG. This data should be written to main memory if any is dirty.
RHSTB	Input	**Read Hit Strobe.** The rising edge of RHSTB results in MS82C443 data entry associated with HADDREG to be latched into RHREG. This data will subsequently be burst to the 486 CPU.

Signal	Type	Description
HP<8:0>	Inputs/ Outputs.	Byte-wide (with parity) data input/output to and from the host CPU.
DW#	Input	**Dirty Word.** Active assertion of this signal qualifies the miss fetch word associated with either SRDYI# or SBRDYI# to be written into the data array of the MS82C443.
HA<3:2>	Inputs	These two bits select a Dword within a given line. These bits are part of the address associated with the data at the host port of the MS82C443's.
HPWEA#, HPWEB#	Inputs	**Host Port Write Enables.** HPWEA# is the host port write enable for bank A and HPWEB# is the host port write enable for bank B. A low assertion of either HPWEA# or HPWEB# indicates the corresponding bus cycle generated by the host is a write cycle. Either of these two signals being sampled low on a clock edge will trigger the latching of HP data into WREG. These two signals cannot be both active simultaneously.
HPOEA#, HPOEB#	Inputs	**Host Port Buffer Output Enables.** HPOEA# will enable data from bank A for the corresponding cycle, while HPOEB# will enable data from bank B. Both signals cannot be active simultaneously. Along with HPOEA# and HPOEB#, SELECT# must be asserted in order to enable the host port outputs.
SP<8:0>	Inputs/ Outputs	Byte-wide (with parity) data input/output to and from main memory.
SA<3:2>	Inputs	These two bits select a Dword within a given line. These address bits are part of the address associated with the data at the system port of the MS82C443's.
SRDYI#	Input	A low activation of this signal indicates that the data read from main memory is ready. The low level of this signal, qualified with a clock edge, triggers the data at the system port data pins to be latched into MUPREG in the burst-RAM at the Dword location by SA<3:2>.
SBRDYI#	Input	Assertion (low) of this signal indicates that data is available on the system port. The low level of this signal, qualified with the clock, latches the system port data pins into MUPREG at the Dword location selected by SA<3:2>.

Signal	Type	Description
SPOE#	Input	**System Port Output Enable.** A low assertion of SPOE# indicates that the system port will be performing a write operation to main memory. For burst writes, SPOE# should be asserted for all four write data transactions. Along with SPOE#, SELECT# must be aserted to enable the system port outputs. For Bypass operations, SPOE# acts as a direction control signal. BYPASS asserted high with SPOE# low creates a host-to-system port bypass, with the contents of WREG being driven onto the system port. BYPASS asserted while SPOE# is high will generate a system-to-host bypass, with system port data being passed directly to the host port.
QWR	Input	**Quad Write.** Activation of Quad Write results in the data residing in MUPREG being written into the MS82C443 data array entry, at the address pointed to by MADDREG and the MS82C443 internal bank select logic. QWR overrides the mux control logic. In addition, QWR resets both the bank select information previously latched through assertion of MALE, and all mask and valid bits associated with MUPREG.
BYPASS	Input	This signal connects the host port data HP<8:0> to system port data pins. Specifically, assertion of BYPASS and SPOE# high connect the system port data pins to the host port. To achieve a host-to-system bypass, BYPASS asserted high and SPOE# asserted low connect the output of WREG to the system port data pins.
SELECT#	Input	**MS82C443 Chip Select.** This signal asserted (low) indicates that the corresponding MS82C443 is selected. A high assertion of this signal will result in all data output pins being tri-stated. For any data outputs to be enabled, SELECT# should be asserted. For complex operations which utilize both the host and system ports, SELECT# should be asserted for the entire operation.
MWB	Input	**Multiple Write-Back.** MWB should be used for write-back cache architectures where each tag entry corresponds to two lines. MWB is asserted during write-back cycles, in the case when the other line associated with the replaced tag is dirty and needs to be written back to memory. The assertion (high) of MWB toggles the A4 bit of the address stored in MADDREG. Subsequent assertion of WBSTB then loads WBREG with the second line of data to be written back to the system.

Section 3. System Block Diagram
i486 CPU System with MS82C441/443

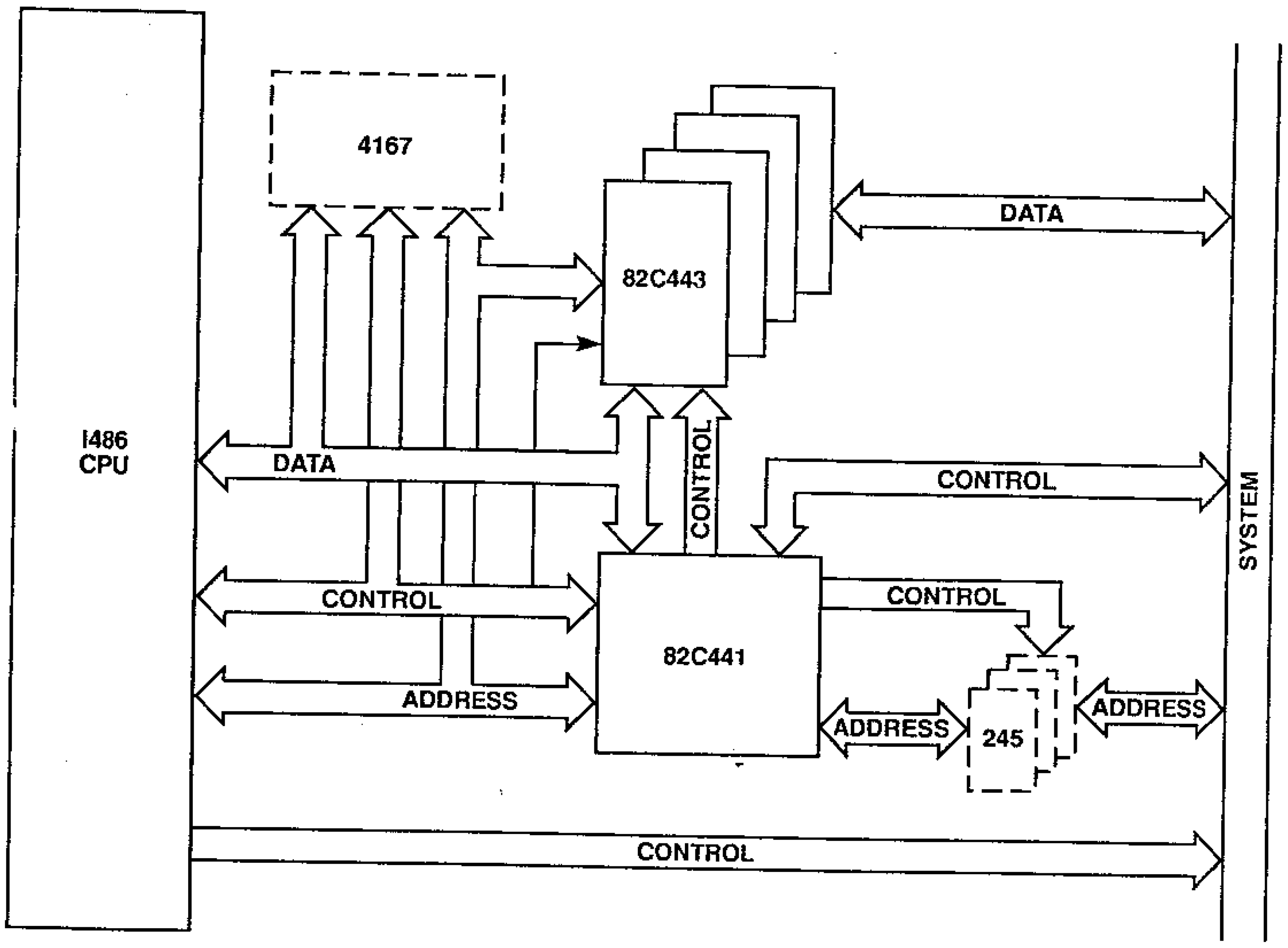


Figure 2. i486 CPU System with MS82C441/443

Section 4. Overview of the MS82C441/443 Architecture

This section will discuss the basic architecture of the MS82C441/443. The discussion will include cache size and organization, system and host bus interface, write back cache operation, and the unique architecture of the MS82C443's.

Cache Size and Organization

The MS82C441, when used with four MS82C443's, supports 64K Bytes of cache memory. The MS82C441 contains all necessary tag arrays, valid and dirty bit array, and control logic to properly operate the cache data arrays contained in the MS82C443's.

The MS82C441 organizes the 64K Bytes of cache into 2048 blocks. Each block is 32 bytes, and consists of two 16-byte lines, or sub-blocks. Each of the 2048 blocks has a 12- or 13-bit tag (depending on cache associativity), and a tag valid bit. Lines are further broken into four Doublewords, or Dwords. There is a valid and dirty bit for each Dword.

Most transfers between system memory, the MS82C443's, and the i486 CPU will be line transfers (four Dword transfers, with each MS82C443 supplying one byte of each Dword). The Dword is the basic unit of transfer in the system, although byte and word operations will occur and can be properly handled by the MS82C441.

The cache data array is stored in four MS82C443's, each containing 16K Bytes of cache memory. The MS82C443's are organized as 2048 x 2 x 4 x 9 (data parity is supported). This can be explained as follows: there are 2048 entries which must be addressed, each entry consisting of two 16-byte lines. Each line is further divided into four doublewords, with each MS82C443 supplying one byte per doubleword. Obviously, each byte (with parity) is 9 bits wide, completing the MS82C443 organization. Refer to Figure 3; the dashed lines show the locations that one MS82C443's cache memory maps into.

Note that each MS82C443 will supply one byte for Dword operations, with one MS82C443 always supplying the same relative byte within a Dword. For example, the MS82C443 logically shown in Figure 3 will always supply the lowest byte (byte 0) of all Dwords in the cache.

Through a bit in the control register, the 2048 blocks can be organized either in direct-mapped mode, or as 2-way set associative. Operation in 2-way set associative mode is normally recommenmded, and will offer higher performance.

Direct Mapped Cache Organization

Organization of the cache in direct-mapped mode is shown in Figure 3. Each physical address that the CPU asserts can only map into one location in the cache. The addres of each cycle driven by the 486 CPU is broken down into several components: the set index, tag, line select, and Dword select. CPU address lines A15:5 (11 bits) compose the set index, and determine which cache location the address can map into. Address lines A27:16 (12 bits) are the tag. Assertion of a bus cycle by the CPU generates a tag comparison, with A27:16 being compared against the tag for the given set index. A match of all 12 bits indicates a cache hit. Address line A4 selects between the two lines of a block. Address lines A3 and A2 select individual Dwords within a line, and are not included in cache hit/miss determinations.

Note that Figure 3 is drawn abstractly for understanding of the cache organization. The tag array bits, tag valid, Dword valid, and Dword dirty bits are contained in the MS82C441, while the actual data for Dwords 0–3 are contained in the MS82C443's.

2-Way Set Associative Cache Organization

Organization of the cache in 2-way set associative mode is shown in Figure 4. As opposed to the direct-mapped strategy just described, each physical address can map into two locations in the cache. As a result, the 2048 entries are mapped into two parallel sets of 1024 entries each.

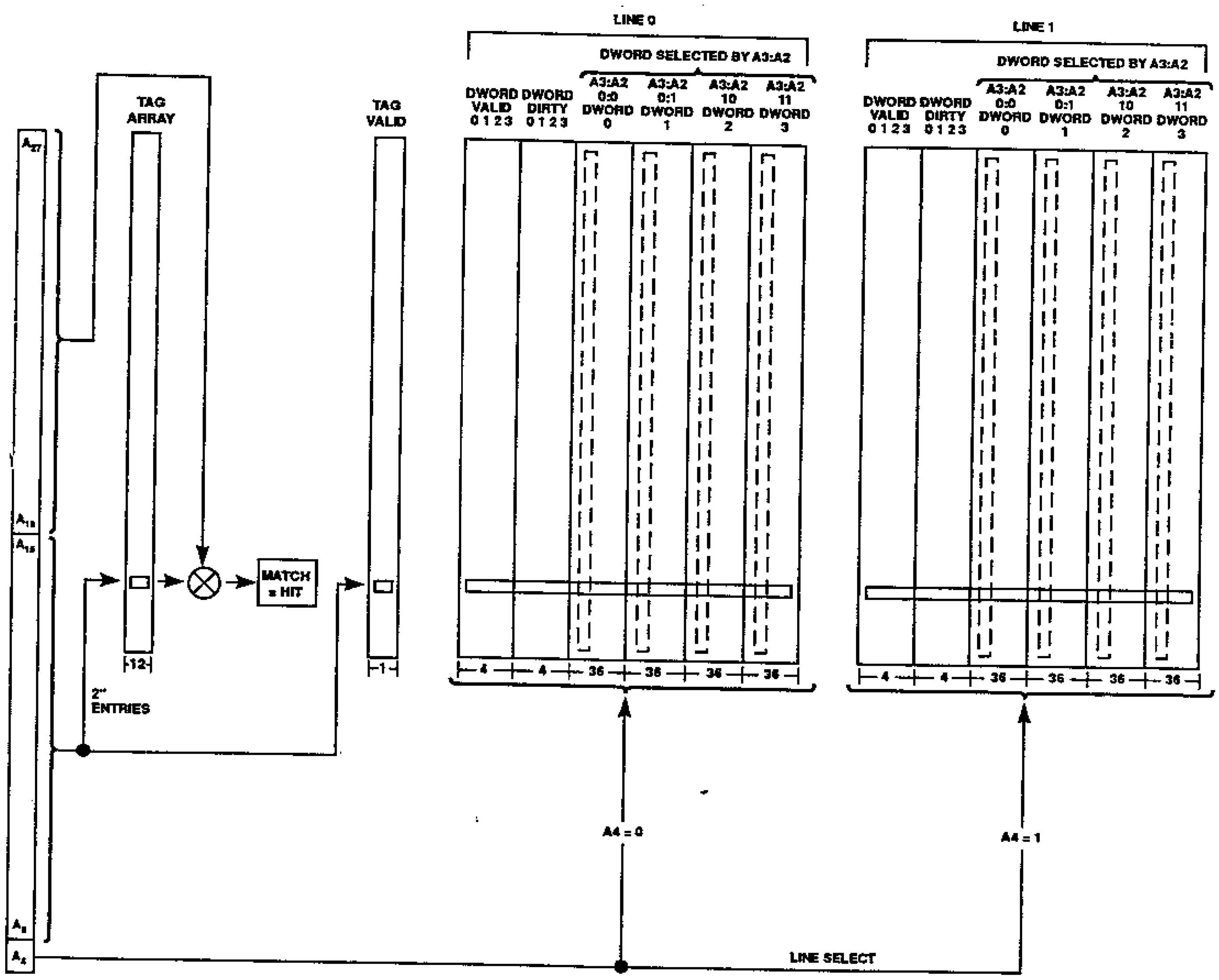


Figure 3. Direct-Mapped Cache Organization

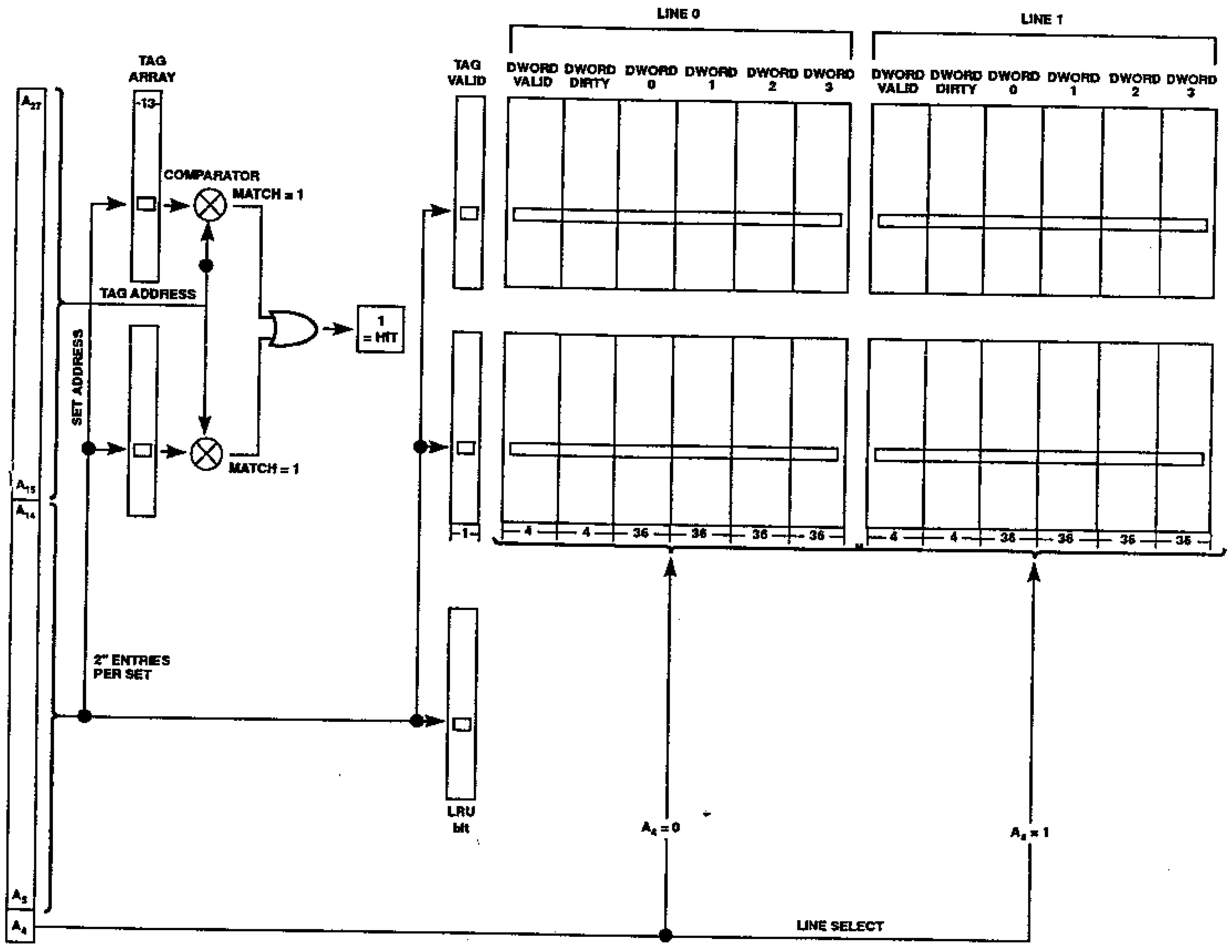


Figure 4. Two-Way Set Associative Cache Organization

Correspondingly, the cache set index is now 10 bits instead of 11, and the tags are 13 bits instead of 12. A bus cycle now triggers two comparison operations, one for each set, at the given set index, with a hit (match) possibly occurring in either set. The results of the comparisons are OR'd together, with a high value from the OR output indicating a tag hit.

System and Host Bus Interfaces

System Interface. For ease of design, the MS82C440 system-side architecture has been designed to appear as much as possible like an i486 CPU to the system. Many of the system-side pins have the same functionality and name as their i486 CPU counterparts. Designers who are familiar with the i486 CPU architecture will find the MS82C441/443 very easy to design with.

Like the 486 CPU, the MS82C443's allow full bursting of reads from the system memory. As an enhancement to 486 CPU functionality, the MS82C441 allows burst quad writes to the system. In addition, critical AC timing parameters have been improved relative to the i486 CPU, to aid designing as much as possible.

Host Interface. In a system, the MS82C441 and MS82C443's reside directly between the CPU and the system logic. This isolation of the CPU allows a local CPU bus and a system bus to be developed. Due to the dual-port architecture of the MS82C443's, processing can occur simultaneously on both of these buses. System cycles, such as buffered writes and write-back cycles, can be performed in parallel with CPU read and write hits on the local bus. The MS82C441 intercepts CPU bus definition signals and addresses, initiating cache operations on those cycles deemed to be cacheable, and passing the remainder on to the system.

On the host side, the MS82C441 has necessarily been designed to follow the i486 CPU local bus. The combination of the MS82C441/443 appear to the CPU as a very fast memory subsystem, completing the great majority of cycles in zero wait states.

Write-Back Cache Operations

Normal cacheable read and write hits will be serviced in zero wait states, with read hits initiating a burst cycle to the CPU, in which a line (four Dwords) will be transferred in five clocks. Write hits will simply update the cache data array, without initiating any system cycles. As 75% of the 486 CPU bus cycles are writes, this offers a primary performance advantage over write-through architectures. Since the data for write hit cycles does not propagate to the system, this data is considered 'dirty' and is so marked by setting the appropriate dirty bit.

Read and write misses, however, necessitate more complicated operations. For best performance, either a read or write miss will generate a system fetch and bring new data into the cache, replacing older data previously present in the cache. However, the data which is replaced may be 'dirty', requiring a write-back cycle to system memory to clear the dirty data. This causes a penalty in traditional single-ported write-back caches, since the system read to fill the cache with new data must wait until the write-back cycle is completed. However, the dual-ported architecture of the MS82C443's eliminates this penalty through the use of hidden write-back cycles which are described in a later section.

Section 5. MS82C443 Architecture Description

The design of the MS82C443's has been optimized to allow high performance for i486 CPU systems. Two main features of the MS82C443's are the optimized data paths and the register set. Figure 5 shows the MS82C443 architecture.

5.1 Array Organization

The MS82C443 interfaces to 80486 systems. It contains a 2 x 2K x 36 array organization. To support an i486 CPU system, four MS82C443's will offer the minimal 64K Byte 2-way associative organization or 64K Byte direct mapping organization. Each MS82C443 contains two nine-bit-wide data ports.

Each MS82C443 contains two banks of 2K x 36 array. Each array is subdivided into four subarrays. For MS82C443's positioned at the most significant byte of the local 32-bit bus, subarray A (2K x 9) will contain the most significant byte of the fourth Dword (highest address) of a quad fetch. Similarly, subarray B (2K x 9) will contain the most significant byte of the third Dword and subarray C will contain the most significant byte of the second Dword.

5.2 MS82C443 Data Paths

The MS82C443 is dual ported. Dual ports allow read/write hit operations to run in parallel with miss processing. It decouples the main memory subsystem from the host local bus through an asynchronous interface using internal holding registers. This allows an easy migration path for system design from 386 to 486 without redesign of the main memory subsystem. Upgrading from a 386 CPU based system to a 486 CPU system becomes purely a local bus issue and is supported by the MS82C441.

The MS82C443 host port supports both scalar reads and writes, and burst reads in 486 CPU systems. For scalar reads and writes, the MS82C443 host port appears just like any standard SRAM. The addition of temporary registers does not induce any additional delay to the scalar read/write path. As a result, the access time for scalar reads will be the same as standard cache data RAMs.

Each MS82C443 offers 32-bit data read and write transfers between the data array and the holding registers. Four MS82C443's in a 32-bit system will offer a true line (16 byte) read and write in one clock. Subsequent access of a burst read through holding registers drastically reduce burst data access time.

The dual port architecture of the MS82C443's allows concurrent processing, or simultaneous processing on both the host and system ports. For example, after a write miss has initiated a system fetch, the MS82C443's may service both the system fetch and any CPU read/write hits that are generated while the system fetch is occurring.

In general, use of the MS82C443's in a system helps performance in four main ways. Namely:

1. On read misses, quad fetches from main memory can occur without waiting for the write-back cycle of the selected replace data. This avoids the historical penalty of write-back caches.

2. Local bus read and write hits can operate in parallel with system bus operations. Write-back cycles will be skipped if none of the data selected for replacement is both valid and dirty.

3. Write-back cycles can be burst to main memory, if the memory is capable of receiving such bursts.

4. And, of course, zero wait burst reads are supported for the 486 CPU at maximum operating frequencies.

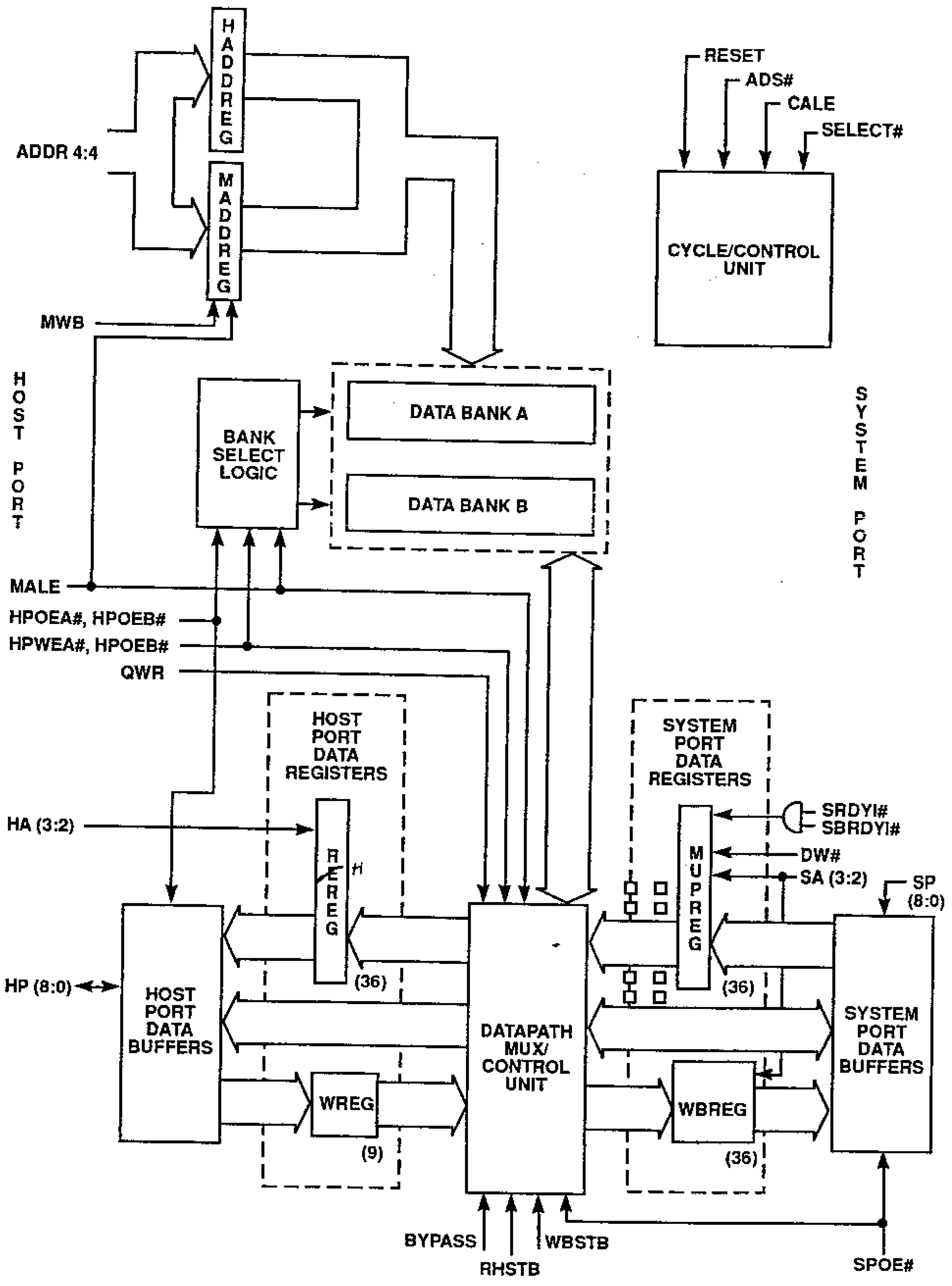


Figure 5. MS82C443 Architecture

6.3 MS82C443 Register Set

To enhance the performance of the MS82C441/443, the MS82C443's contain special holding registers. The intent of these registers is to retain the performance advantage that write-back caches enjoy for write hits, while eliminating the penalty that traditional write-back cache architectures sustain during read/write miss replacement cycles.

The six registers are:

Address Registers:

HADDREG — Hit Address Register
MADDREG — Miss Address Register

Data Registers:

WREG — Write Register
RHREG — Read Hold Register
WBREG — Write Back Register
MUPREG — Memory Update Register

Address Registers

The MS82C443 contains two address registers. HADDREG is an 11-bit register which contains the hit address and MADDREG is also 11 bits, containing the miss address. The address from the CPU will be latched into HADDREG register in the MS82C443 through activation of ADS#. The activation of ADS# will open HADDREG latch to the address decoder of the data array in the MS82C443.

MADDREG is used during miss cycles to latch the initial miss address. This address is later used during miss processing, so that data retrieved from the system can be correctly updated into the MS82C443 data array. The address in HADDREG will be latched into the MADDREG register through the activation of MALE.

Data Registers

The WREG is a 9-bit register. All writes to the array from the host port propagate through the WREG. In addition, WREG can be used as a buffer on write cycles, for either buffered write-through cycles or buffered non-cacheable writes. For example, writes to addresses defined as non-cacheable addresses are buffered in WREG and consequently can be completed in zero wait states on the local bus. These writes continue on the system side until the system accepts the written data. Meanwhile, local bus operations may continue (see dual port architecture discussion below).

The RHREG is a 36-bit register, used to allow one clock burst reads from the MS82C443's. Essentially, during the first transfer of a burst read, 32 bits of data are read into RHREG. To complete the second, third and fourth transfers, the contents of RHREG are driven onto the local CPU bus, one Dword at a time.

RHREG is used to support local host bus (486) burst reads. A burst read hit will cause all four 32-bit doublewords within the same line to be read into RHREG. "Wrapped-around" burst order is supported. The first demand word will be fetched and sent to the host port directly from the MS82C443 data array. The subsequent three remaining words are now fetched from RHREG instead of through the data array. This architecture allows very high speed (50 MHz and beyond) burst mode possible without using ultra-fast (sub 10 ns) data RAMs. Burst-order is controlled by HA<3:2> and is totally transparent to the MS82C443.

The WBREG is a 36-bit register used to hide the write-back cycles that occasionally occur when system data from read misses will replace dirty data. The older line, containing the dirty data, is latched into WBREG, allowing the system read to occur with no delay. At the completion of the system fetch, the contents of WBREG are then written back to the system. For best performance, the MS82C441 allows these quad writes to be burst to the system, if the system memory can accept such bursts.

For miss operations in write-back mode, WBREG holds data from the selected data line to be replaced and write back to main memory if the replaced line contains valid and dirty data. This allows burst writes on the system memory port SP<7:0> without binding the data array for write-back replacement cycles. This arrangement allows the cache data array to serve the host port for local bus read and write hits.

MUPREG is a 36-bit register, used as a holding register for incoming data from system quad fetches due to read/write miss cycles. As each Dword is returned by the system, it is passed on to the CPU and latched inside MUPREG. At the completion of the system fetch, all four Dwords are written into the cache data array by assertion of the quad write (QWR) signal.

For optimum performance, the incoming data from the system is not loaded directly into the cache data array. Instead, MUPREG will be used to hold the data fetched from main memory. MUPREG is a 32-bit register, and can contain data for up to four transfers. Inclusion of MUPREG in the MS82C443 architecture allows systems to gain a performance benefit. Since incoming system port data is latched into MUPREG instead of the data array, the array is free to service local bus hit operations as the read miss processing occurs on the system port.

MUPREG contains quad fetch miss data from main memory. MUPREG contains the entire line of quad fetch data. The order of loading the MUPREG register is controlled through SA<3:2>, making burst-order to main memory purely transparent to the burst-RAM. Once the line is loaded into MUPREG, the entire line is loaded into the cache data array in one clock by activating the QWR signal.

MUPREG Organization

Associated with each byte of MUPREG is both a valid and a mask bit.

The functionality of the mask and valid bits is shown below. When QWR is asserted (high) on a clock edge, the contents of MUPREG will be updated into the data array, as pointed to by MADDREG and the previously latched bank select information. Each byte of MUPREG will be written to the array if its valid bit is set (indicating valid data from the system port), and its MASK bit is cleared (indicating no advance write occurred for that byte).

Condition	On QWR and CLK:
MASK# & VALID#	NO WRITE
MASK# & VALID	WRITE OCCURS
MASK & VALID#	NO WRITE
MASK & VALID	NO WRITE

Figure 6 shows MUPREG organization.

Mask bits are set during advance writes, by assertion of MALE#, SELECT#, and either HPWEA# or HPWEB#. The mask bit set within MUPREG is selected by SA<3:2>. Setting of the mask bits will be further described in the section concerning Advance Writes. The valid bits are set by assertion of SELECT# and either SRDYI# or

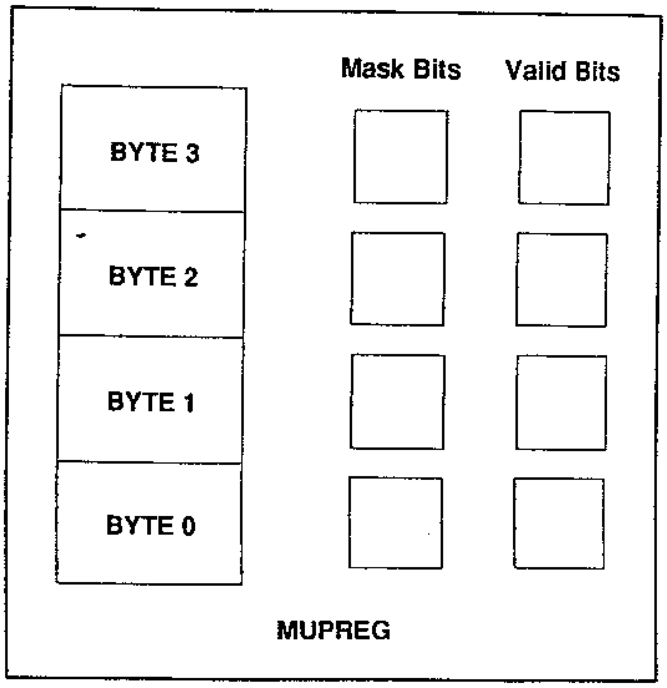


Figure 6

SBRDYI#. As with the mask bits, the valid bit set within MUPREG is selected by SA<3:2>. A further discussion of the valid bits is in the section on System Port Writes.

Assertion of QWR (quad write) clears all valid and mask bits associated with MUPREG. This allows the next advance write or system fetch to begin.

6.4 MS82C443 Bank Select

Each MS82C443 contains two banks of 2K x 36 data array.

Reads and writes on the MS82C443 host port directly use the bank select pins. For example, host port reads from and writes to bank A are selected by HPOEA# and HPWEA#, respectively.

A low assertion of HPWEA# signifies a write cycle from the host port to bank A through WREG, and a low assertion of HPWEB# signifies a write cycle from the host port to bank B through WREG. HPOEA# and HPOEB# are active low output enables. A low assertion of HPOEA# will gate the read data from bank A to host port and the low assertion of HPOEB# will gate the read data from bank B to the host port. HPWEA#, HPWEB#, HPOEA# and HPOEB# cannot be active simultaneously.

However, the MS82C443 contains no corresponding system port bank select pins. For system port reads and writes, the MS82C443's use previously-latched inputs for bank selection.

Detection of a CPU miss, indicated by assertion of MALE (Miss Address Latch Enable), will latch bank select information. HPOEA# or HPWEA# asserted indictes a bank A operation, while HPOEB# or HPWEB# indicates bank B. After MALE, this bank select information is remembered by the MS82C443. Subsequent system port read (array to WBREG) and write (MUPREG to array) operations are directed to the bank previously selected when MALE was asserted. QWR (quad write) activation or assertion of RESET will clear the latched bank select information.

Similarly, the SELECT# (MS82C443 chip select) signal is latched through MALE for the system port miss processing operations between the MS82C443's and main memory.

Section 6. MS82C443 Functional Description

Table 1, below, summarizes control of the MS82C443 Burst-RAMs.

Condition	System Port Data Comes From:
SPOE# and BYPASS# and SELECT#	WBREG
SPOE# and BYPASS and SELECT#	WREG
Condition	**Host Port Data Comes From:**
SELECT# and SPOE and BYPASS	System Port
When SELECT# and BYPASS are low, and either HPOEA# or HPOEB# low and:	**Host Port Data Comes From:**
RHSTB# and WBSTB#	Data Array
RHSTB and WBSTB#	RHREG
WBSTB	MUPREG

ADS# or CALE:

Falling edge makes HADDREG flow-through.
Rising edge latches address into HADDREG.

MALE:

Inhibits write operations to cache data array.
HADDREG address latched into MADDREG.
Latches bank select information for future SP operations.

QWR:

Contents of MUPREG written into data array at MADDREG.
MUPREG mask and valid bits cleared.

WBSTB:

Latches data at address in MADDREG into WBREG.

HPWEA# or HPWEB#

The falling edge of either of these makes WREG flow-through.
Either low on a clock (MALE low) latches HP into WREG.
Activates bank select information.

BURST-RAM OPERATIONS

6.1 Host Port Operations

6.1.1 Host Port Single Reads

MS82C443's allow single read operations with the local bus processor through the host port. A single read operation is shown in Figure 7.

The falling edge of ADS# will activate the HADDREG (hit address register) into flow-through mode. Activation of SELECT# enables the internal bank select logic of the MS82C443's. The falling edge of HPOEA# or HPOEB# indicates to the MS82C443 which bank will supply the data. The data array is accessed and read to the local processor through the host port. After access times t9 (valid address to valid data delay), t18 (data valid delay from HPOEx#), and t19 (data valid from HA<3:2>) have passed, valid data is available on the host port of the MS82C443's.

6.1.2 Host Port Burst Reads

The architecture of the MS82C443's supports burst mode read operations. Each MS82C443 contains an internal 32-bit Read Hold Register (RHREG) to facilitate high-speed burst read hit operations. Up to four-transfer burst are possible with the architecture of the MS82C443. Figure 8 shows a four-transfer burst operation.

The first transfer of a burst read is accessed similarly to the scalar read previously described. ADS# assertion will activate HADDREG into a 'flow-through' mode. RHSTB should be deasserted (low) to allow data from the cache array to bypass RHREG and be sent to the host port data pins.

After the completion of the first transfer, RHSTB can be asserted. This has two effects. First, the entry in the data array pointed to by the HADDREG and bank select inputs will be latched into RHREG as the rising edge of RHSTB occurs. Second, RHSTB being held high (while BYPASS and WBSTB are low) will connect the output of RHREG to the host port data pins.

During the access that RHREG is loaded, two timing parameters must be met in order to supply valid data on the host port. Both tx (RHSTB to valid HP data) and ty (HA<3:2>) to valid host port data) must be met. After both of these parameters have been satisfied, RHREG drives valid data onto the host port.

Once RHREG has been loaded, the remaining transfers of the burst read will come from RHREG, making very fast burst accesses possible. Burst order will be controlled by the HA<3:2> inputs, with the burst order transparent to the MS82C443's. As HA<3:2> toggles to the next burst-order address, valid data from RHREG will be available after ty has passed. RHSTB should be held high and WBSTB low to keep the RHREG connected to the host port.

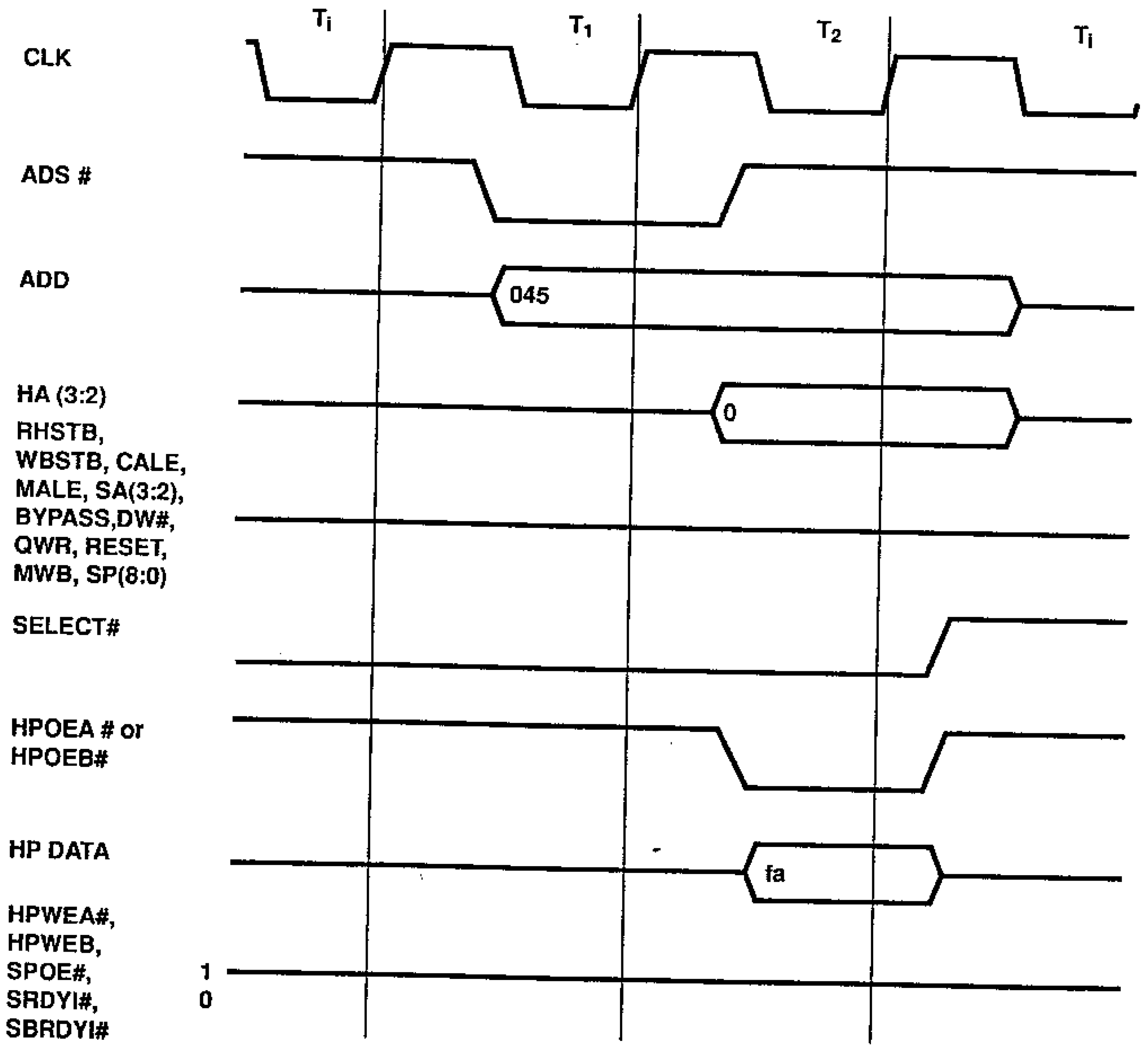
Note that once RHREG has been loaded, the data array is available for system port operations. In addition, the host port is available for use while miss processing is occurring on the system side.

6.1.3 Host Port Single Writes

The MS82C443 supports single host port write operations. Burst writes are not supported by the MS82C443.

The falling edge of ADS# sets HADDREG into 'flow-through' mode. Similarly, assertion of either of the two write enables (HPWEA# or HPWEB#) will cause host port data to begin flowing through the WREG, as well as select which of the two banks of the MS82C443 data array is to be updated. HPWEA# and HPWEB# can trigger the write operation. Sampling either of these inputs asserted (low) on a rising clock edge, with MALE deasserted, will result in the data in WREG being written into the MS82C443 data array. HA<3:2> will control which byte within a Dword in the MS82C443 data array is to be updated.

Figure 9 shows a host port write operation to the MS82C443.



T_i
T_1
T_2
T_i
CLK
ADS #
ADD
045
HA (3:2)
0
RHSTB, WBSTB, CALE, MALE, SA(3:2), BYPASS,DW#, QWR, RESET, MWB, SP(8:0)
SELECT#
HPOEA # or HPOEB#
HP DATA
fa
HPWEA#, HPWEB, SPOE#, SRDYI#, SBRDYI#
1
0

Figure 7. Host Port Single Read

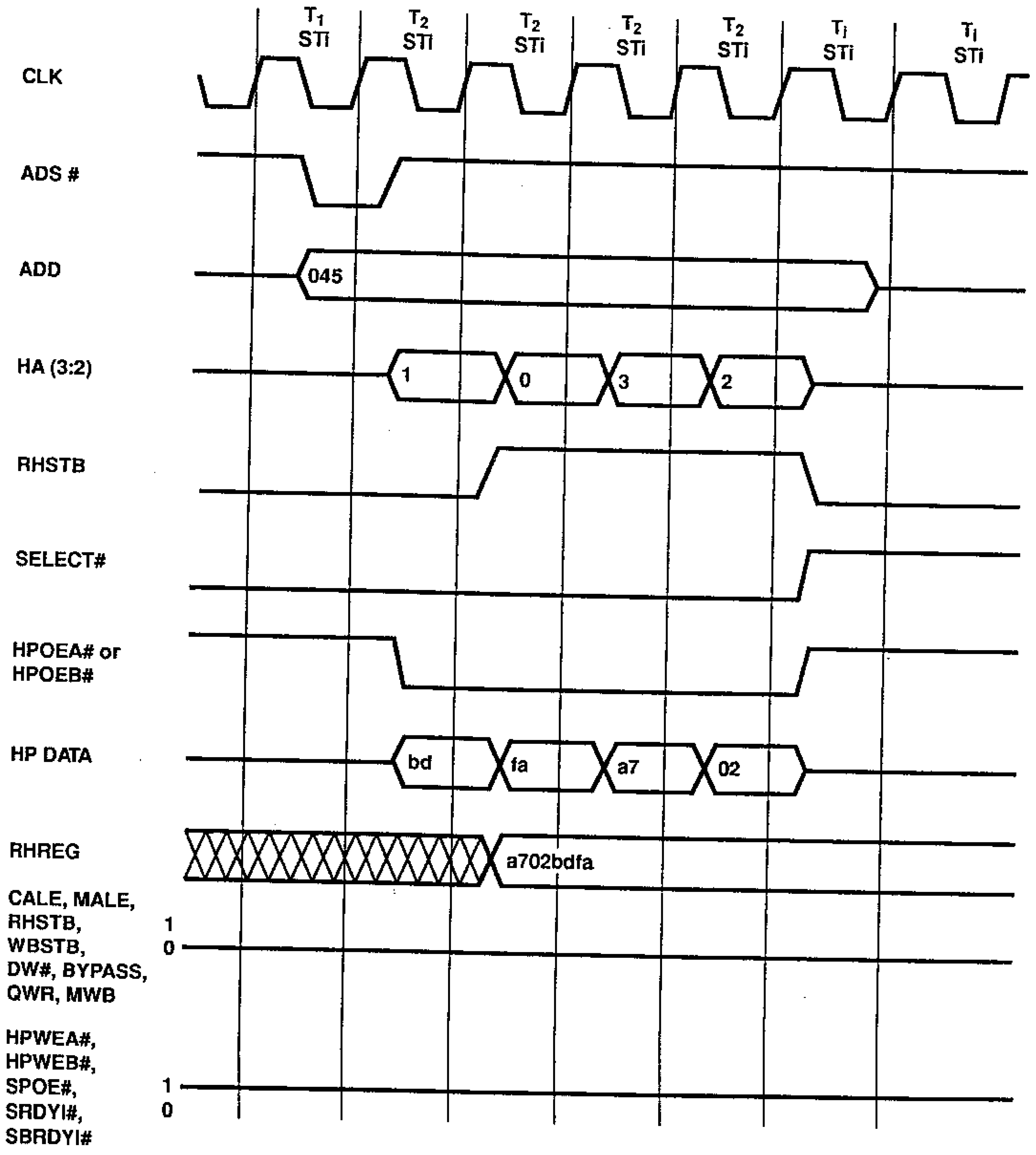


Figure 8. Host Port Burst Read

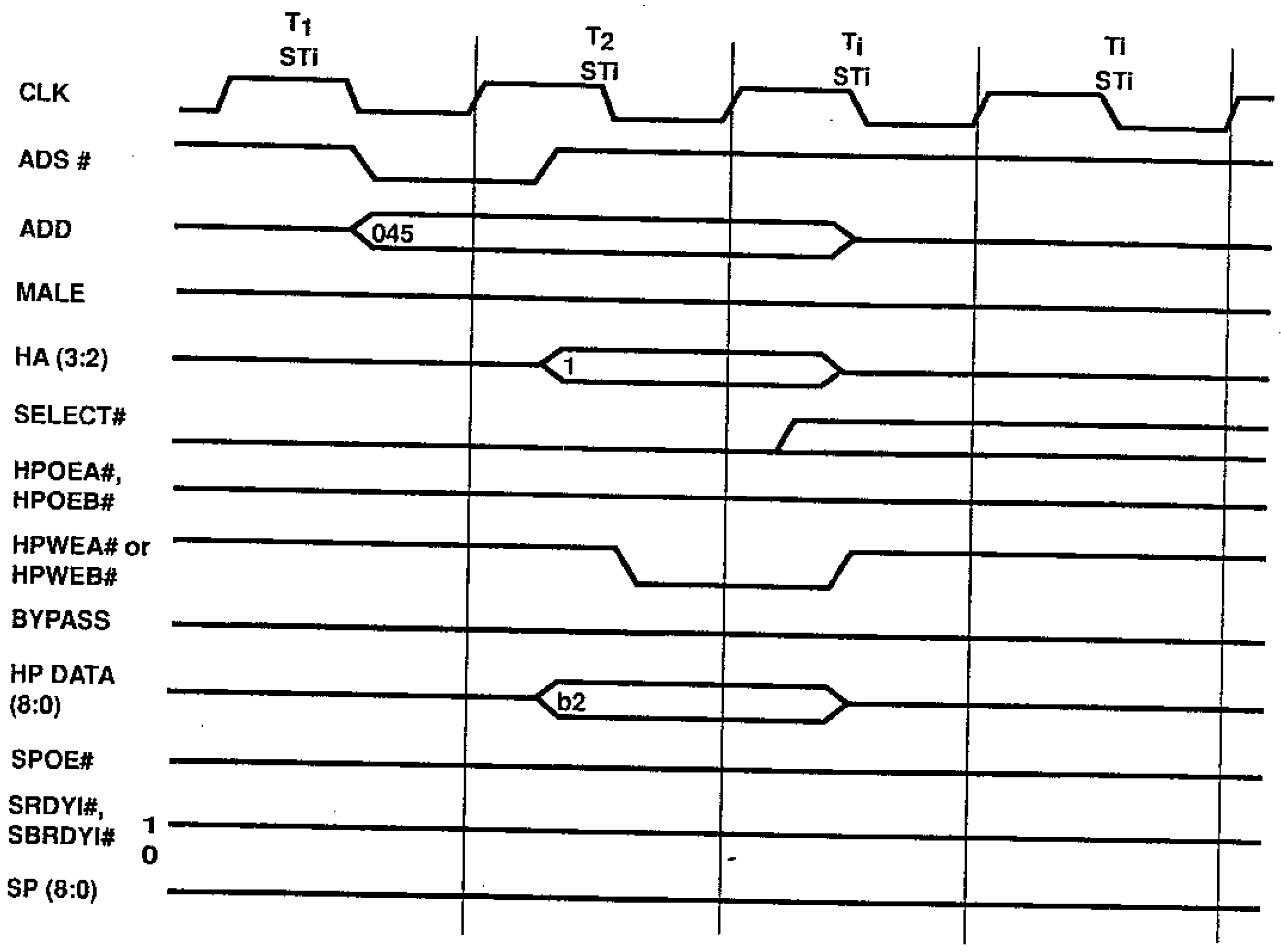


Figure 9. Host Port Single Write

Activation of MALE will inhibit WREG data from being written into the data array. The write enable signals are sampled on every clock edge. Hence, a write into the data array will occur only on the rising edge of a clock, if MALE is inactive and either HPWEA# or HPWEB# asserted. As will be shown later, buffered writes can be performed by asserting HPWEA# or HPWEB# to latch data into WREG, and then inhibiting the array write by asserting MALE.

Finally, as the write to the data array occurs, the corresponding bit in MUPREG will be set. This allows 'advance' writes to occur. See section 7.3.2 for a detailed description.

6.2 System Port Operations

6.2.1 System Port Single Reads

There is no direct path between the data array and the system port. Cycles which require the burst-RAMs to supply data on the system port (such as snoop reads) may be accomplished as follows. First, the assertion of either ADS# or CALE allows the address to flow through HADDREG. MALE should be asserted to latch the request into MADDREG. The assertion of WBSTB and HPOEA# or HPOEB# latch data into WBREG. Finally, SPOE# asserted, while BYPASS is deasserted, will enable the contents of WBREG onto the system port. SA<3:2> will select among the four bytes of WBREG. For read cycles, the states of SRDYI# and SBRDYI# are not used. Figure 10 details a single read operation on the system port.

6.2.2 System Port Burst Reads

Burst reads from the array to the system port may be accomplished. One case where burst reads from the array to the system port are necessary are for flushing the cache. These reads would begin as detailed above in the single read case. Once WBREG has been loaded with four bytes of data, these bytes may be burst onto the system bus byte-by-byte, with SA<3:2> toggling to select among the four bytes. SPOE# should remain asserted, and BYPASS deasserted in order to enable the contents of WBREG onto the system port.

7.2.3 System Port Single Writes

As before in the system port read case, there is no path to allow direct writing from the system port to the data array. Writes from the system port must first propagate through the MUPREG before they can be stored in the array. Snoop writes are an example of system port write cycles.

Before MUPREG can be written into the data array, the bank of the MS82C443 which is to be written to must be selected. This is done as described before, by assertion (low) of SELECT# and either the HPOEA# or HPOEB# inputs. The assertion of MALE latches both the bank select information and the address into MADDREG.

MUPREG is loaded by the assertion of either SRDYI# or SBRDYI# on a rising clock edge. The byte of data appearing on the system port is latched into MUPREG, as selected by the SA<3:2> inputs.

As the 9 bits of data from the system port are updated into MUPREG, the corresponding valid bit will be set if DW# is inactive (high). This valid bit being set will allow the corresponding byte of MUPREG to be written into the MS82C443 data array, when QWR is later asserted. The presence of valid bits allow abortion of quad fetch on read miss for a subsequent pending write request in write-through mode to access the main memory interface. Valid bits make partial write of a fetched line possible.

Activation of the DW# signal indicates that the MS82C443 data array contains dirty data at the same address. Correspondingly, activation (low) of DW# inhibits dirty and valid miss data from being overwritten by quad fetch miss data by clearing the corresponding valid bit.

Once MADDREG is loaded and the bank selected, MUPREG may be written into the array. This write is accomplished by assertion of QWR on a clock edge. The bytes of MUPREG which did not receive any writes from the system port will not be updated into the array, as the valid bit for these bytes is cleared.

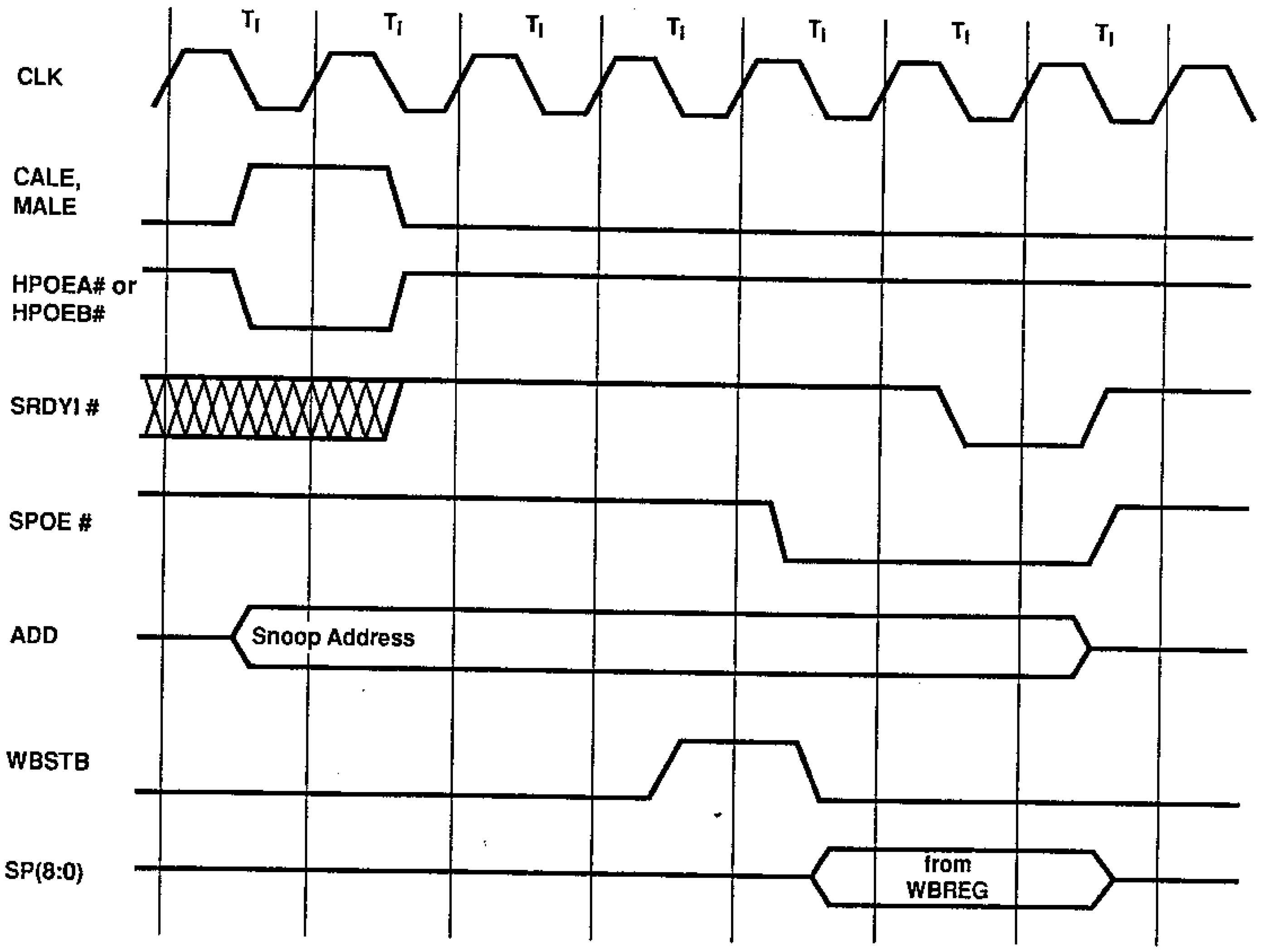


Figure 10. System Port Single Read

The quad write operation clears all valid and mask bits associated with MUPREG.

Figure 11 shows a single write operation to the array through the system port.

6.2.4 System Port Burst Writes

System port burst write operations execute as above described for system port single writes. As MUPREG is a 32-bit register, up to four bytes may be loaded into MUPREG before its contents are written into the data array. Assertion of SRDYI# or SBRDYI# on a clock edge loads the system port data into the MUPREG byte pointed to by SA<3:2>. The four bytes of MUPREG can be loaded as quickly as four clocks.

6.3 Dual-Port Operations

6.3.1 Bypass Operations

For optimum performance, the architecture of the MS82C443's allows bypass operations. Bypass can occur in either direction.

6.3.1.1 Host to System Port Bypass

Some host bus cycles may be designated as write-through cycles. The architecture of the MS82C443's supports these cycles.

Activation of BYPASS together with SPOE# (low) will create a host-to-system bypass condition, with data from the WREG being passed directly to the system port. The assertion of BYPASS and SPOE# will override and reset the selection information previously latched by the rising edge of MALE. The device controlling the MS82C443's must ensure that all miss processing operations are completed at the system port before any bypass write cycles begin.

Updates to the cache data array may occur during bypass operations as previously described in the section on Host Port Writes. The combination of either HPWEA# or HPWEB# asserted, SELECT# asserted, and MALE negated on a rising clock edge will generate a write into the cache data array during a host-to-system port bypass.

6.3.1.2 Buffered Host to System Port Bypass

Write operations to the MS82C443's may occur as buffered writes. As described above, the falling edge of either HPWEA# or HPWEB# allows host port data to begin flowing through the WREG. Once WREG has been loaded with valid data, the buffered write may then be accomplished by asserting BYPASS high and SPOE# low. The contents of WREG will be driven onto the system port data pins while these two inputs remain asserted.

Since buffering occurs in the WREG, buffered write operations may occur whether or not an update into the cache data array occurs.

Figure 12 shows a cache update due to a host port write, with the write being buffered and continuing on the system port, until the system accepts the write data. Buffered write misses will be detailed in the write miss section; however, they proceed identically except for the MALE input. Unlike the BYPASS read case, in BYPASS writes, the state of the MALE input is recognized.

Figure 13 details a buffered write operation, where no cache update occurs. MALE is asserted to inhibit the write operation.

6.3.1.3 System to Host Port Bypass

System-to-host port bypasses may also be generated. During cache read miss cycles, the requested data may be bypassed asynchronously from the host port to the system port in order to minimize the miss penalty and optimize performance. Use of the BYPASS path allows read miss processing to occur as quickly as possible, with no clock latencies between arrival of incoming data at the system port and forwarding of the same data on to the host port. Designers should allow for the BYPASS propagation delay from the system port to the host port, in addition to the the normal CPU read data setup time.

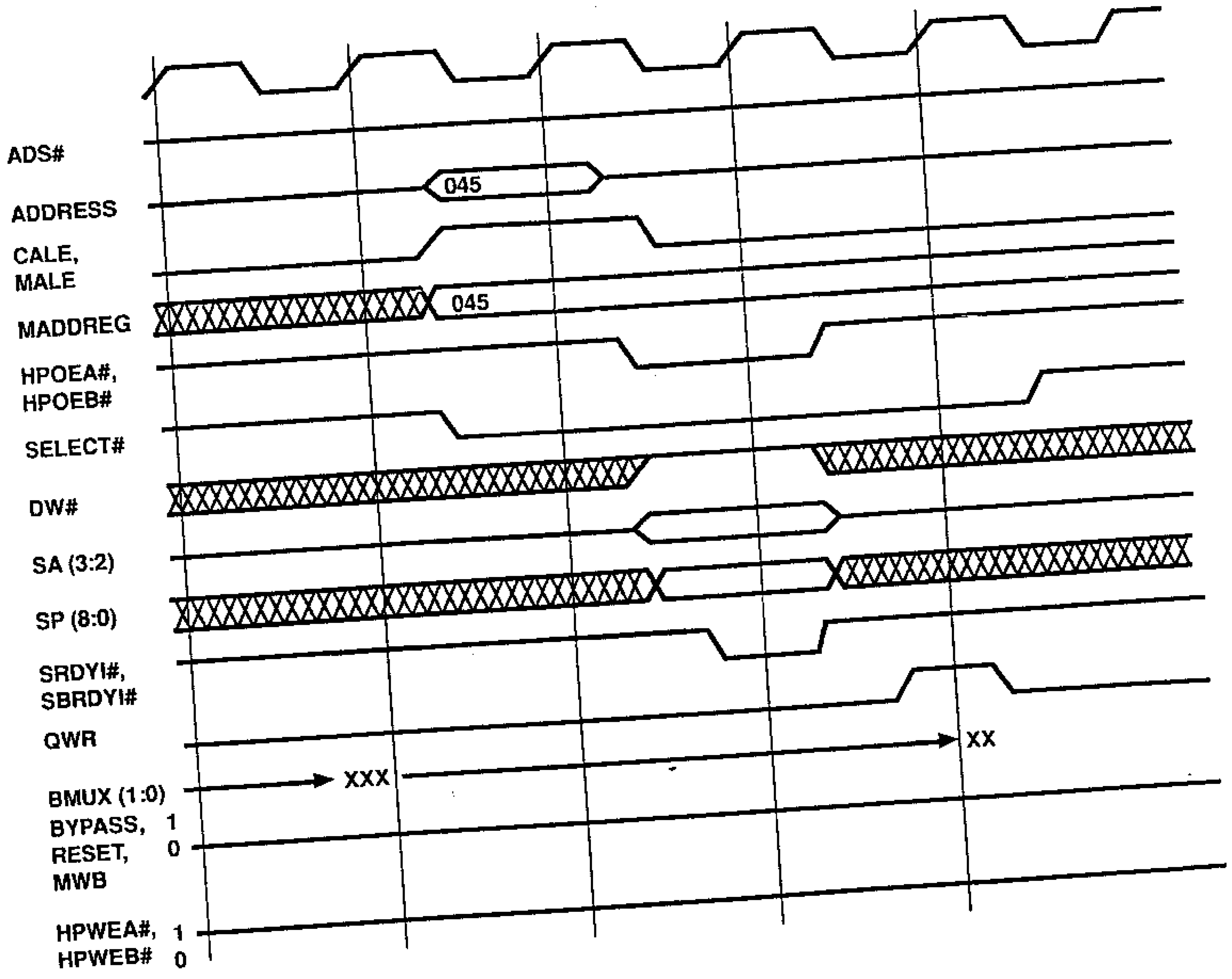


Figure 11. System Port Single Write

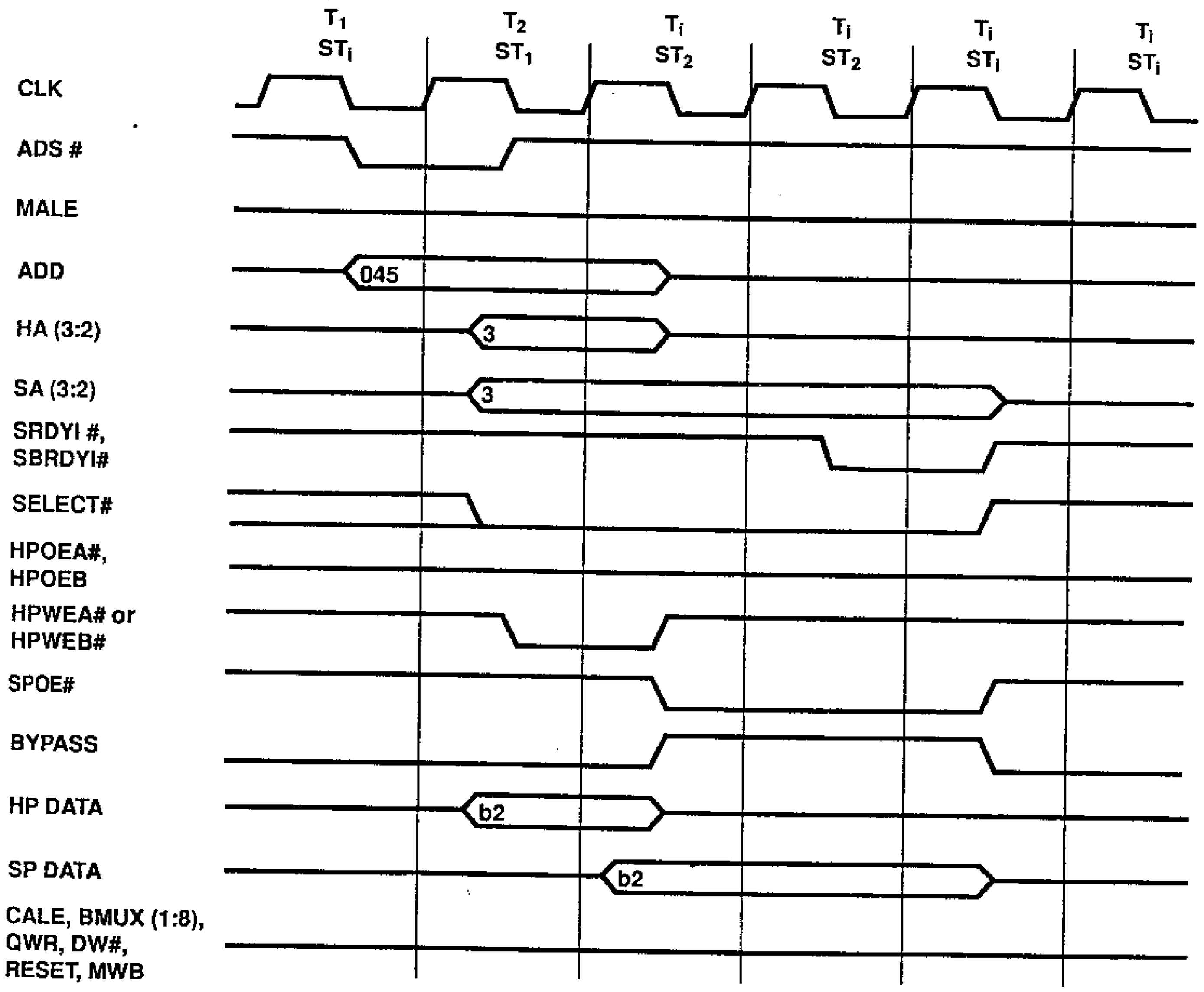


Figure 12. Buffered Host-to-System Bypass with Update

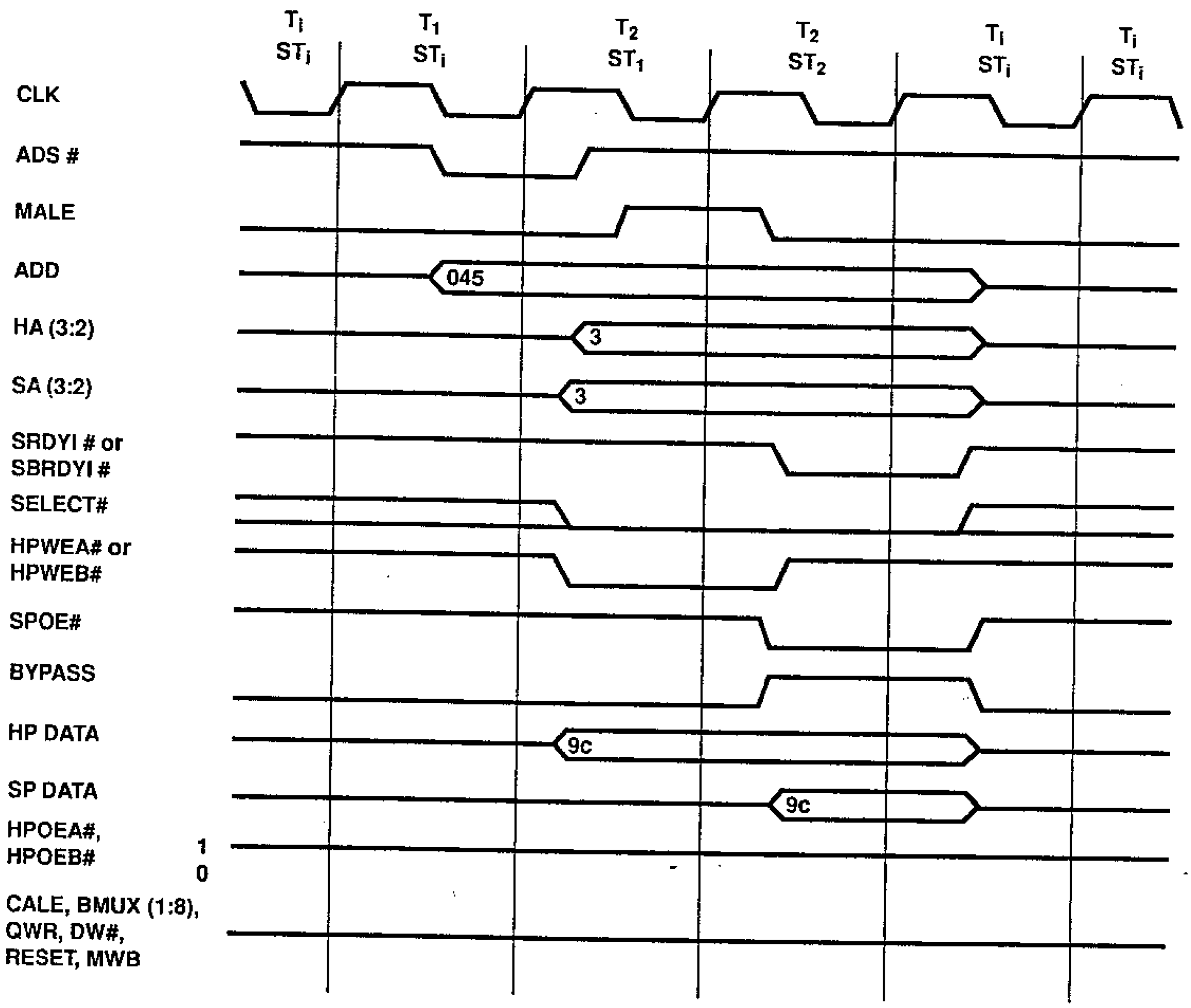


Figure 13. Buffered Host-to-System Bypass Without Update

As the data arrives at the system port, it may be bypassed directly to the host port by assertion of the BYPASS and SPOE# signals high. Note the dual functionality of SPOE#. When BYPASS is deasserted, SPOE# is used to enable system port data from WBREG onto the system port. When BYPASS is asserted, however, SPOE# acts as a direction control for BYPASS.

SRDYI# or SBRDYI# sampled asserted (low) on a rising clock edge will latch system port data into MUPREG. Since MUPREG is a 36-bit register, SA<3:2> will select one of four bytes in MUPREG.

In addition, cycles which the cache controller designates as non-cacheable may be easily handled with the BYPASS signal. By asserting BYPASS, the requested data will be supplied by the system port rather than the cache data array. Figure 14 shows a non-cacheable cycle, with the assertion of BYPASS and SPOE# held high generating a system-to-host bypass. Note that when BYPASS is asserted, MALE becomes a "don't care" input.

As the system-to-host bypass cycle occurs, system port data may be latched into MUPREG. This can be accomplished on rising clock edges by assertion (low) of either the SRDYI# or SBRDYI# inputs. The MS82C443 does not differentiate between these inputs, and they are internally ANDed together. Assertion of either of these on a clock edge will latch system port data into one byte of MUPREG, as selected by SA<3:2>, provided data has been valid one setup time previous to the rising edge of the clock.

Assertion of the QWR signal (on a clock edge) will then write the contents of MUPREG into the cache data array. Any mask bits set in MUPREG will inhibit writes to the corresponding byte in the data array.

6.3.1.4 System to Host Port Bypass with Reordering

MUPREG can act as a buffer register if the burst-order between the 486 host and main memory is different. Memory subsystems using DRAM nibble mode are likely to use sequential burst order, unlike the i486 CPU. As each of the subsequent three Dwords of the burst are read from main memory, they are bypassed to the host port if the burst orders are the same. A differential in burst order will result in data coming from MUPREG. QWR can be asserted once all miss data from memory is updated into the MUPREG register.

Figure 15 details a read miss, where reordering occurs between the system and host ports.

6.3.1.5 System to Host Port Bypass with Partially Dirty Lines

Some cache architectures may contain lines of data which are partially dirty. Figure 16 shows a line which is partially valid and partially dirty. For i486 CPU reads from such a line, the MS82C443 must supply the data which is dirty, while the system must supply the portion of the line which is not present in the cache. The architecture of the MS82C443 supports such situations.

System-to-host port bypass cycles may be interrupted by negation of the Bypass signal. When Bypass is negated, the MS82C443's will supply data as selected by either HPOEA# or HPOEB#, the host port address, and HA<3:2>. Figure 17 shows a CPU read miss, from the line detailed in Figure 16. During the third transfer, the negation of Bypass causes the MS82C443's to supply the dirty data from its array.

6.3.2 Advance Writes

The MS82C443 architecture supports "advance" host port writes in write-back cache architectures. "Advance" write miss processing means that write miss data can be directly updated into the data array in the MS82C443. System fetches at the same address, in order to fill the remainder of the cache line, can occur subsequently and be written into the array without overwriting the previously stored data.

There is a mask bit for each byte of MUPREG. Mask bits can be used to support advance writes. As a write miss (SELECT# and either HPWEA# or HPWEB#) from the host port (through WREG) occurs, the mask for the corresponding byte (as selected by HA<3:2>) in MUPREG is set. Any subsequent fetches from the system port will not overwrite the data now written into the array.

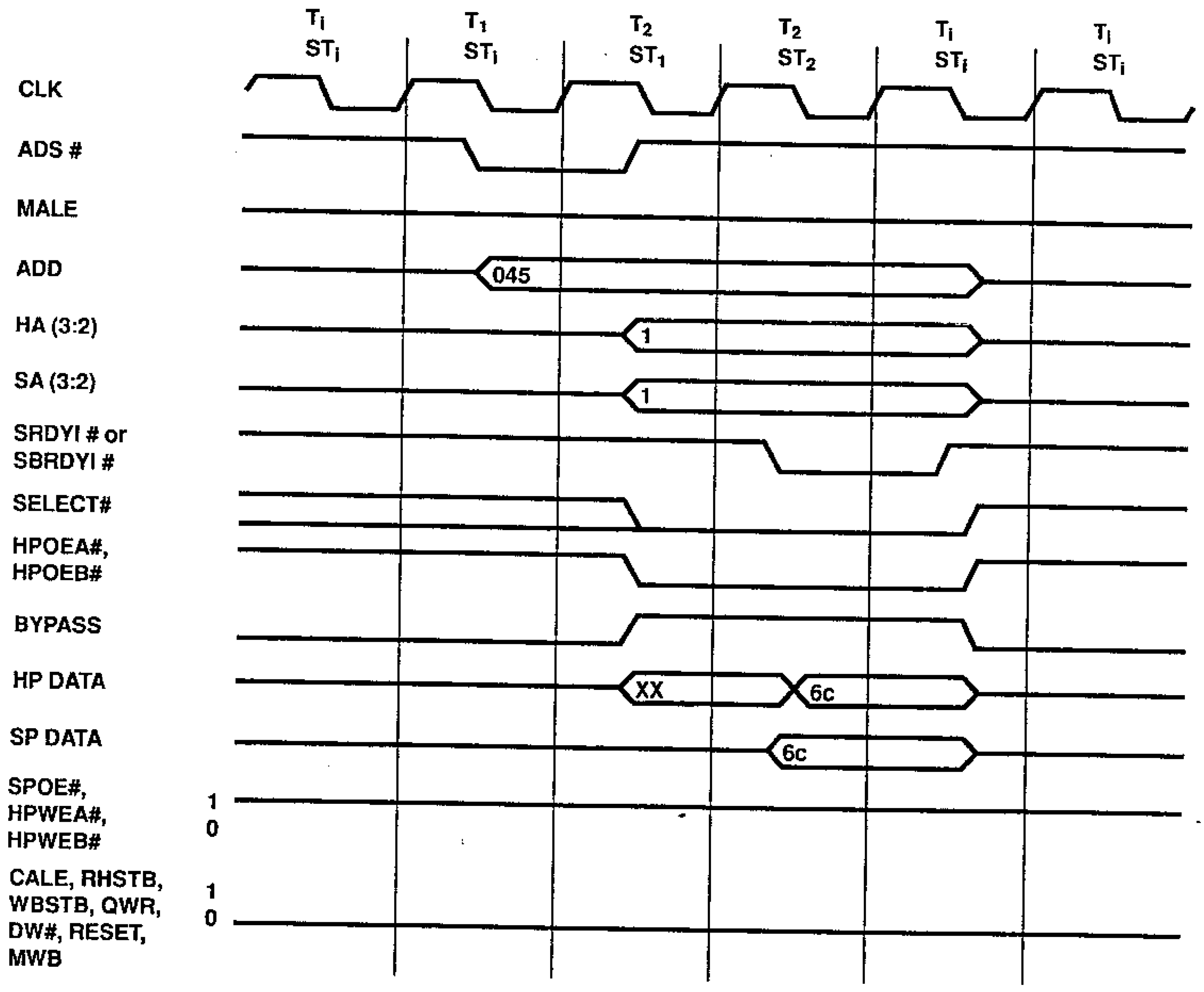


Figure 14. System-to-Host Port Bypass

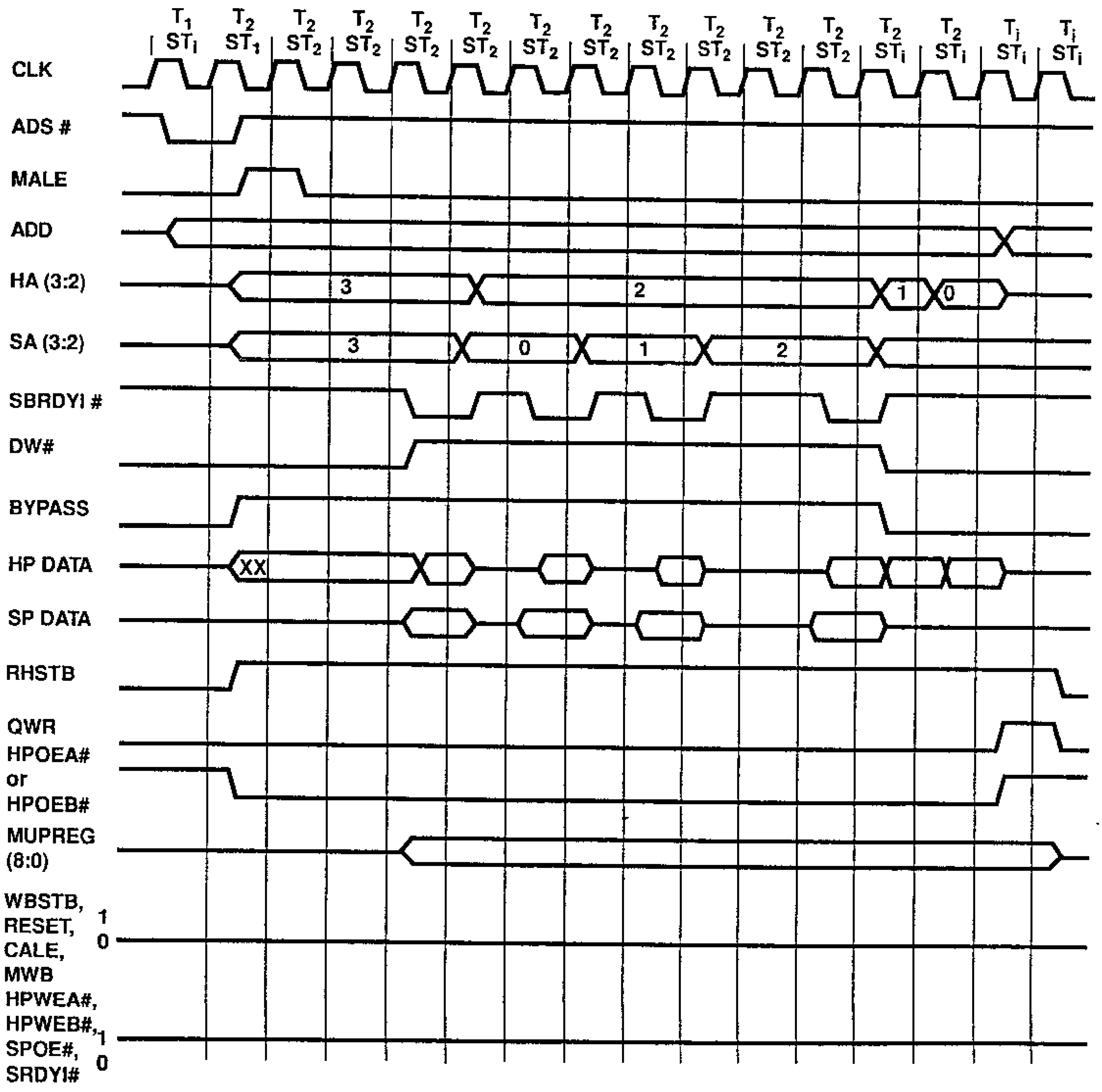


Figure 15. System-to-Host Port Bypass with Reordering

03	02	01	00
48	07	66	2c
Valid & Dirty	Non-Valid	Non-Valid	Non-Valid

Figure 16. Example Cash Line Used in Figure 17

For example, consider a host port write to the data array of the least significant byte (HA<3:2>=0). On the rising edge of the clock that the write is triggered in the array (HPWEA# or HPWEB# is asserted), the mask bit of MUPREG also corresponding to the least significant byte (SA<3:2>=0) is set. A system fetch can then occur to retrieve the remainder of the line (improving the cache hit rate), filling MUPREG as the fetch occurs. When the contents of MUPREG are written to the data array by assertion of QWR, any MUPREG bytes with set masks will be protected from overwrite.

At reset, all four mask bits associated with each byte of the MUPREG are cleared.

Figure 18 shows an advance write occurring.

6.3.3 Evacuation

CPU miss cycles which bring new data into the cache from the system port will usually replace valid data in the cache. For write-back cache architectures, the replaced data may be dirty. In these cases, the dirty data must be evacuated from the cache data array and written to system memory, so as not to be simply overwritten by the incoming data from the system port.

The architecture of the MS82C443's allows for easy evacuation of dirty data into the WBREG. In addition, additional performance is obtainable, since read miss processing can occur at full speed, with WBREG being written to the system after the system fetch completes.

6.3.3.1 Host Port Read Miss with Evacuation

Figure 19 details a CPU read miss cycle, necessitating a system-to-host port bypass operation with evacuation. ADS# signals the beginning of the cycle. MALE latches bank select information and the address into MADDREG. Later assertion of WBSTB latches the data which is to be replaced into WBREG. Incoming data from the system port is forwarded directly on to the host port through assertion of Bypass. SPOE# is held high to correctly enable the Bypass direction, and directly connect the system port to the host port. At the end of the system port cycle, QWR writes the data in MUPREG into the data array. Finally, the contents of WBREG must be written to the system port.

6.3.3.2 Host Port Advance Write with Evacuation

The rising edge of WBSTB will trigger the latching of data (selected by MADDREG and previously latched bank select information) into WBREG. A write-back burst sequence should occur if the data replaced is dirty.

The falling edge of ADS# signals the beginning of the cycle. HPWEA# and HPWEB# are asserted in order to select the bank to be written. However, before the write can occur, the dirty data must first be evacuated from the data array. As such, MALE should be asserted along with HPWEA# or HPWEB# to inhibit the write operation from occurring. The rising edge of WBSTB will trigger the latching of the data from the selected replace line into WBREG. A write-back cycle should occur if the data replaced is dirty.

As in the read miss case, MUPREG will be used to hold quad fetch data from main memory. Miss data will be fetched in a "wrapped-around" fashion with the demand word fetched first. Each byte of the MUPREG is associated with a valid bit. As each byte is updated into MUPREG through SRDYI# assertion, the corresponding valid bit will be set if DW# is active. This valid bit will qualify the corresponding

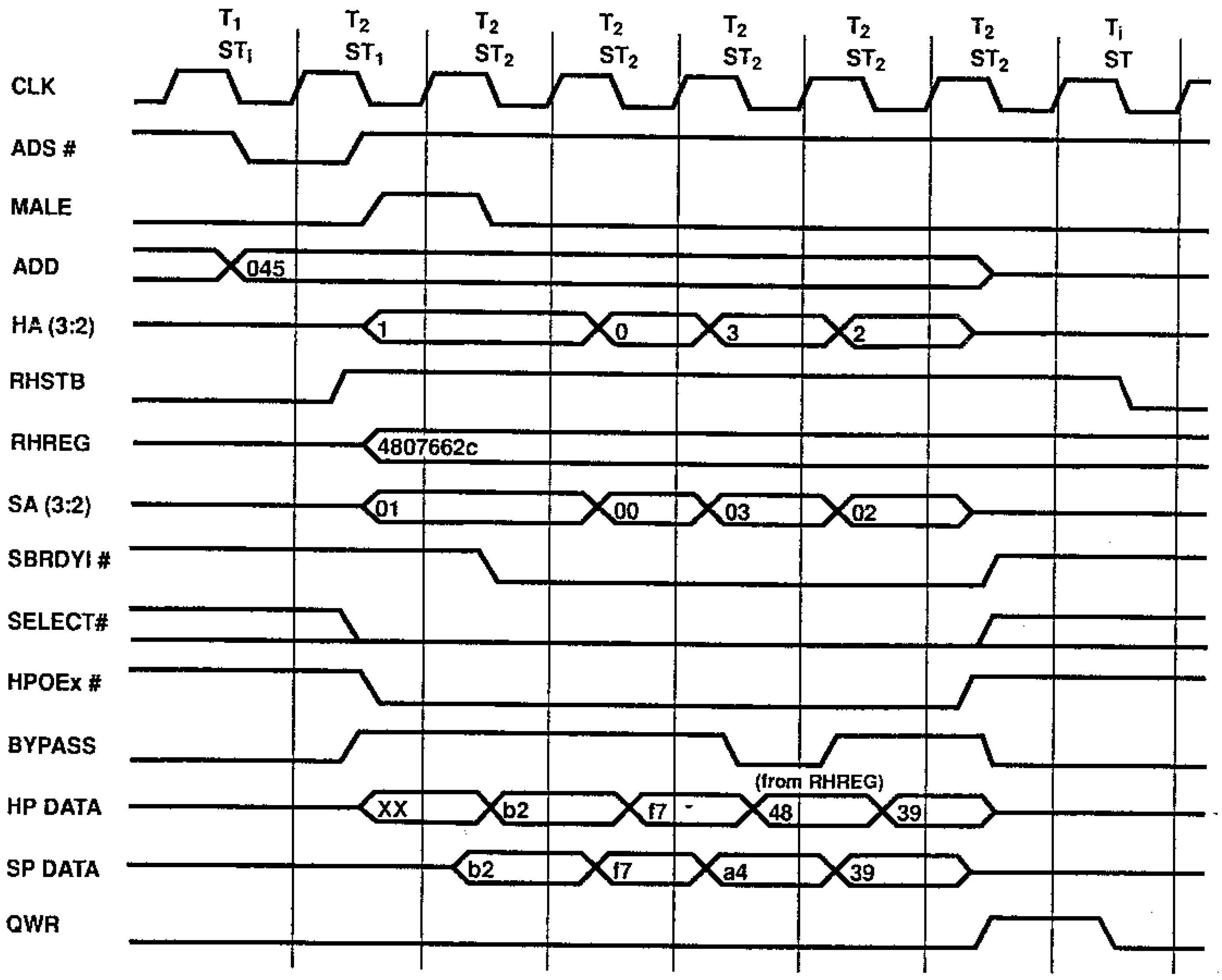


Figure 17. System-to-Host Port Bypass with Update and Partially Dirty Line

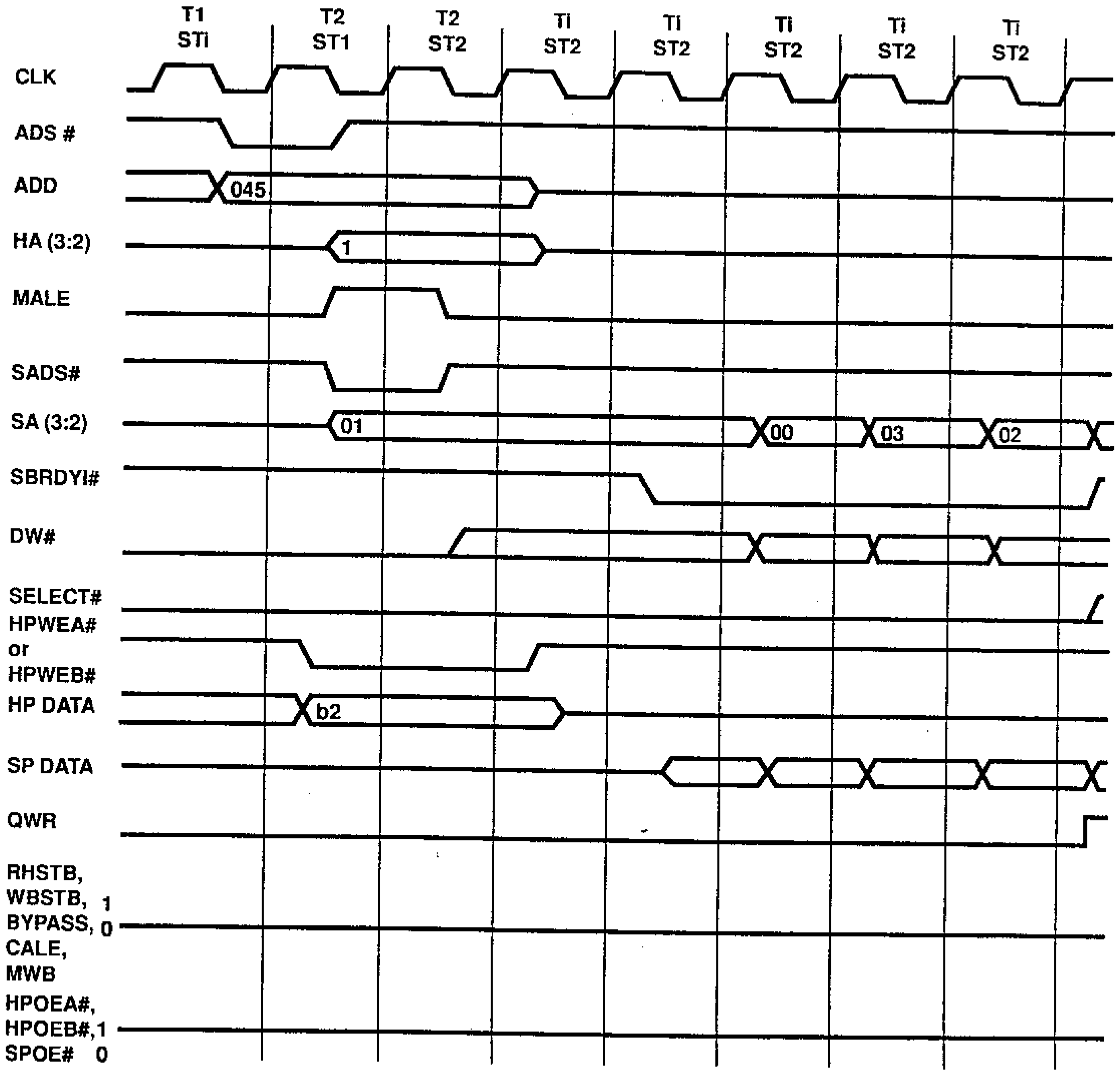


Figure 18. Advance Write and Subsequent Quad-Fetch

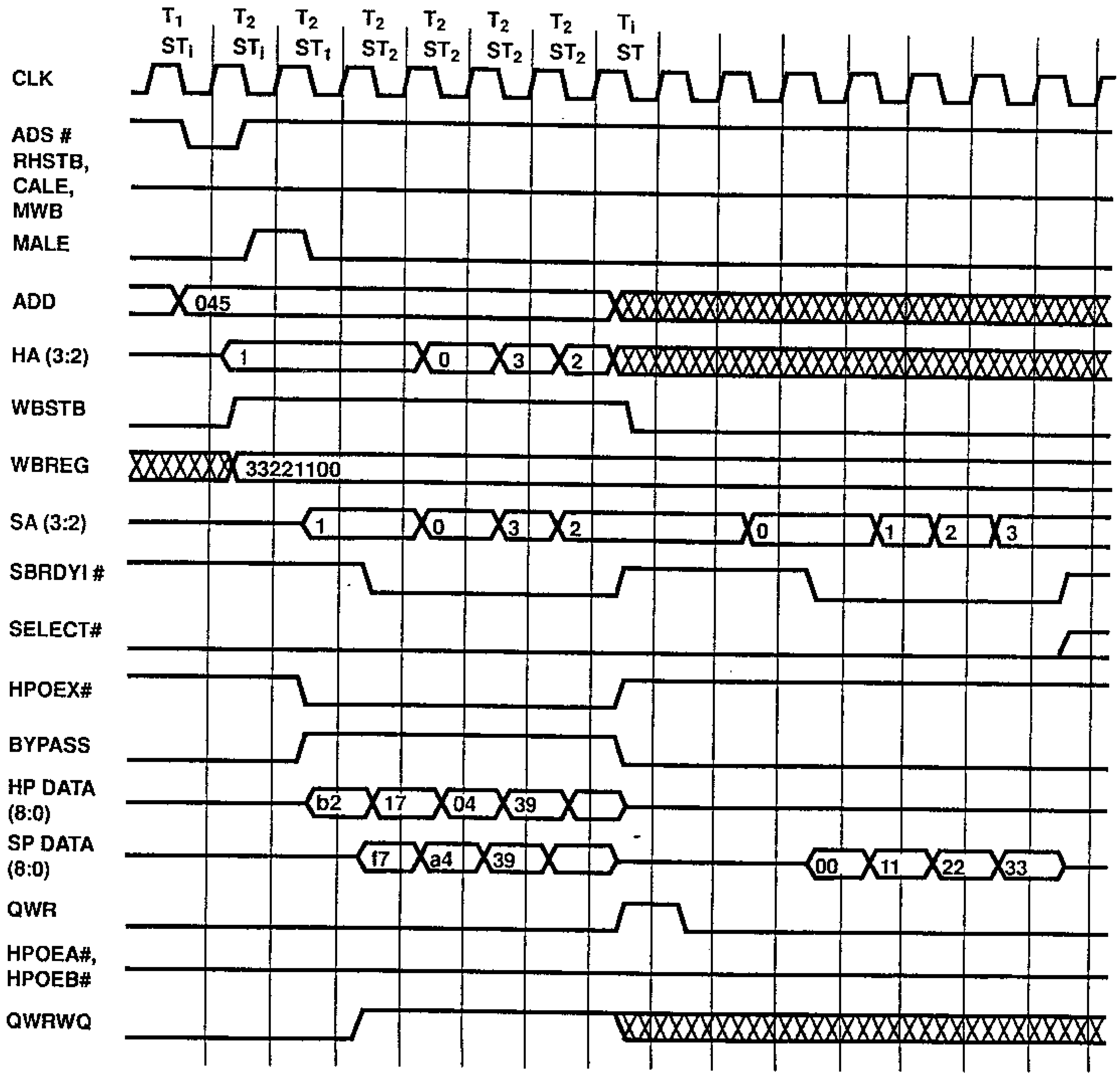


Figure 19. Read Tag Miss with 1 Write-Back

byte of the MUPREG to be written into the MS82C443 data array as QWR is asserted. After each QWR (auad write) cycle, all valid and mask bits associated with MUPREG will be reset.

Figure 20 details a write miss, with evacuation of dirty replaced data and the ensuing system quad fetch. At the end of the quad fetch, the contents of WBREG are written to memory.

Figure 21 shows the same write miss cycle as before; however, at the end of the quad fetch, MWB is asserted in order to toggle MADDREG to point to the second line belonging to the replaced tag entry. After MWB is asserted, WBSTB loads WBREG with the contents of this line. Finally, a write-back cycle to the system of this newly loaded data occurs.

6.3.4 Concurrent Processing

The dual port architecture of the MS82C443 is one of its most powerful features. The MS82C443 is capable of processing on the system and host ports concurrently.

There are several instances where concurrent processing is possible. First, buffered write-through cycles can occur through the WREG. As the write continues on the system bus, the host port can process read and write hits. Also, system read and write requests can occur in parallel with host port operations. And, as has been previously shown, write-backs of dirty data can occur from WBREG and be hidden from the CPU. While these write-backs occur, local CPU cycles can be satisfied on the host port.

Figure 22 shows many of the features of the MS82C443 in use simultaneously. A CPU write miss occurs on the host port. This miss will be written into the data array through an 'advance' write. The dirty data in the array which is to be evacuated is loaded into WBREG. To increase the hit rate of the cache, a system quad fetch occurs to fill the remainder of the line that the CPU write updated. This system fetch is transparent to the CPU, and will be stored in the data array without overwriting the 'advance' write. While this fetch completes, the CPU generates another write cycle and a burst read cycle, which are both satisfied on the host port.

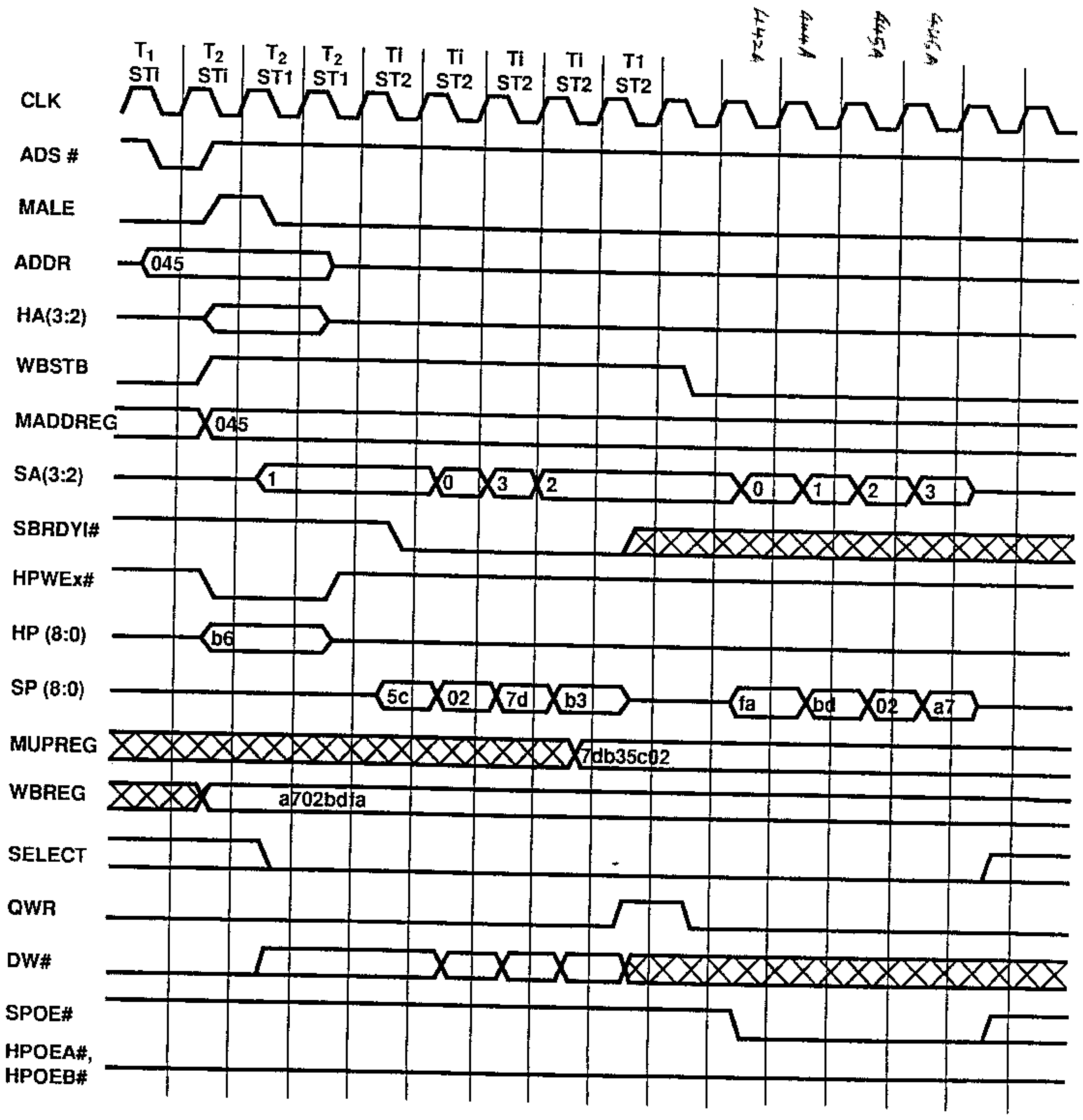


Figure 20. Advance Write with Subsequent Quad Fetch and 1 Write-Back

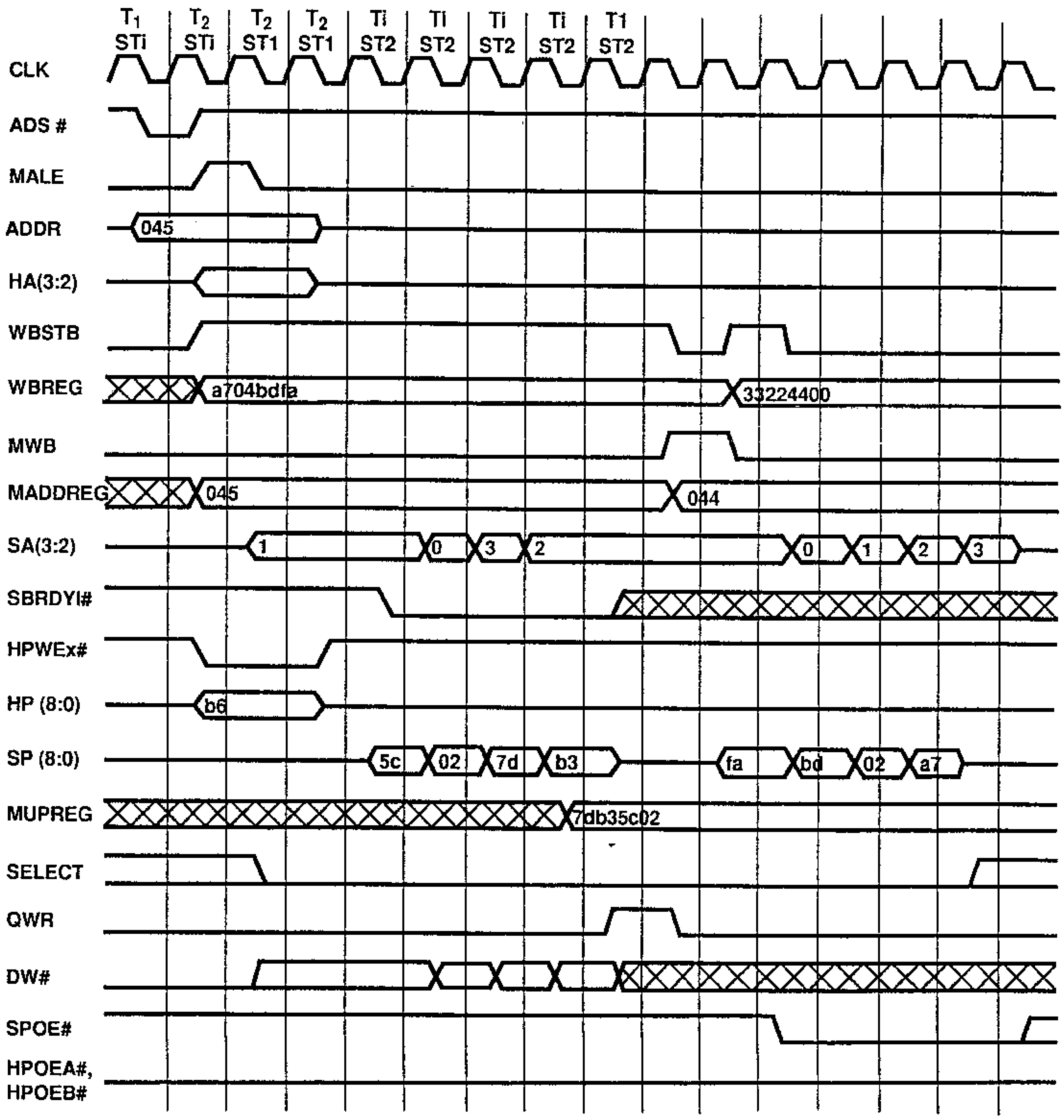
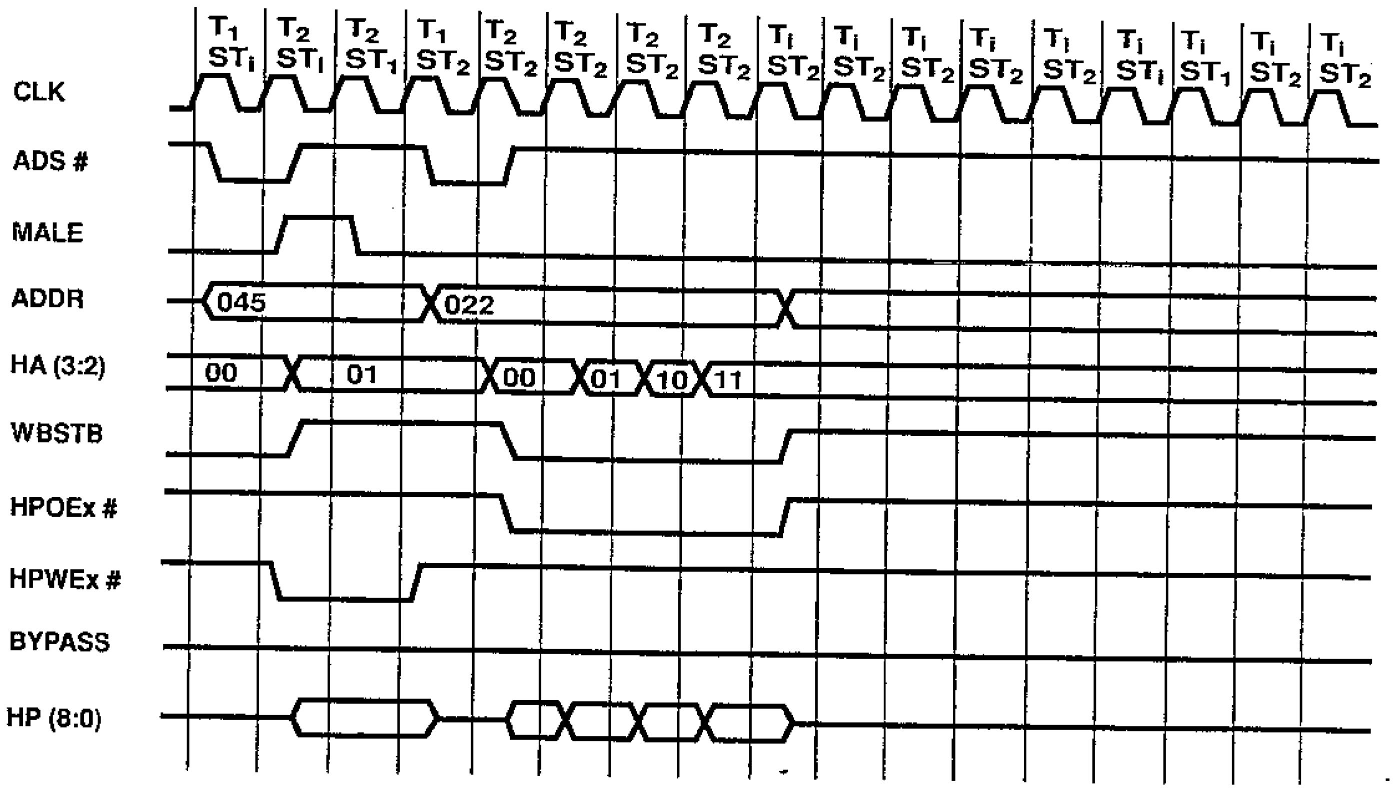


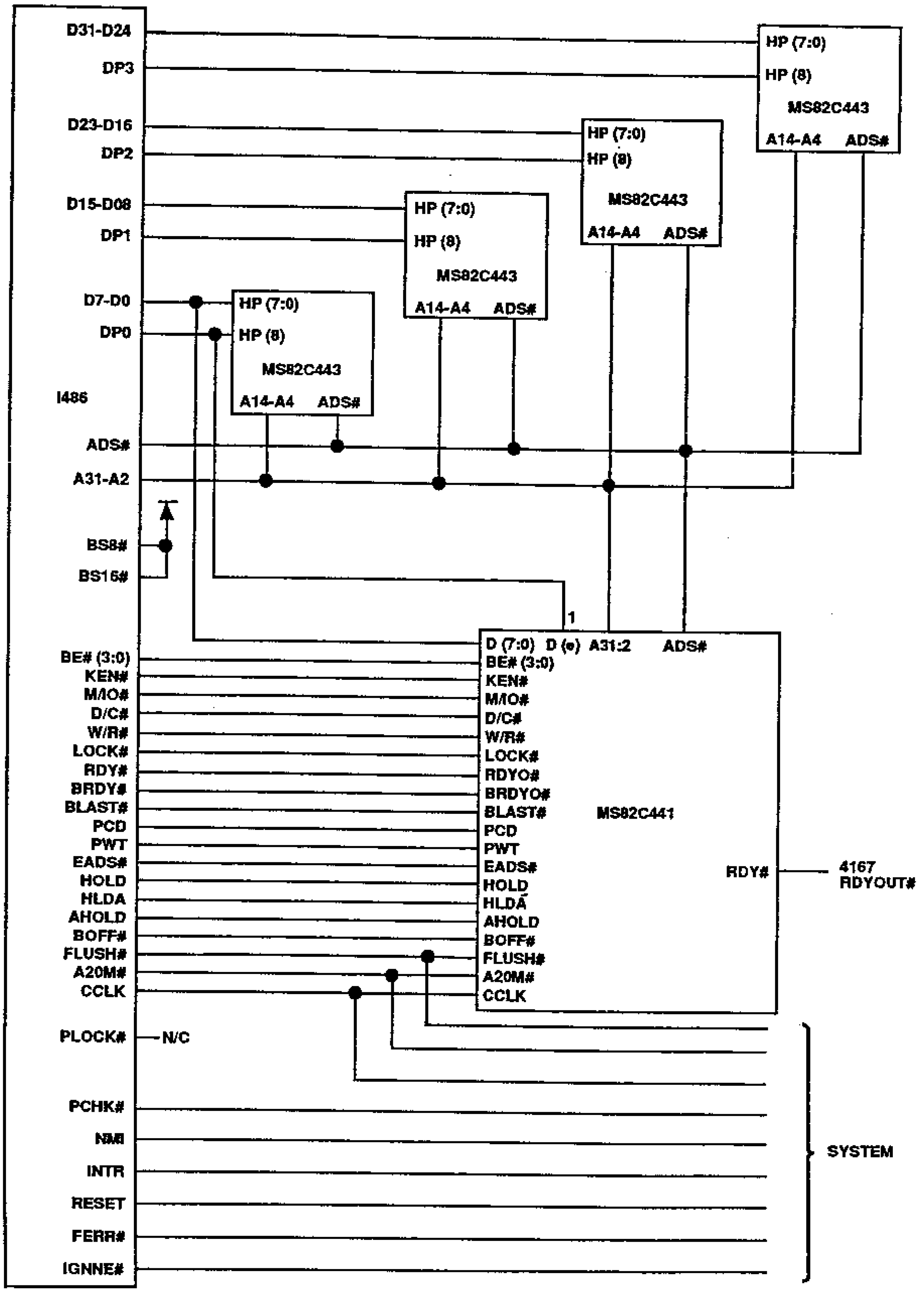
Figure 21. Advance Write with Subsequent Quad Fetch and 1 Write-Back (Neighboring Line)



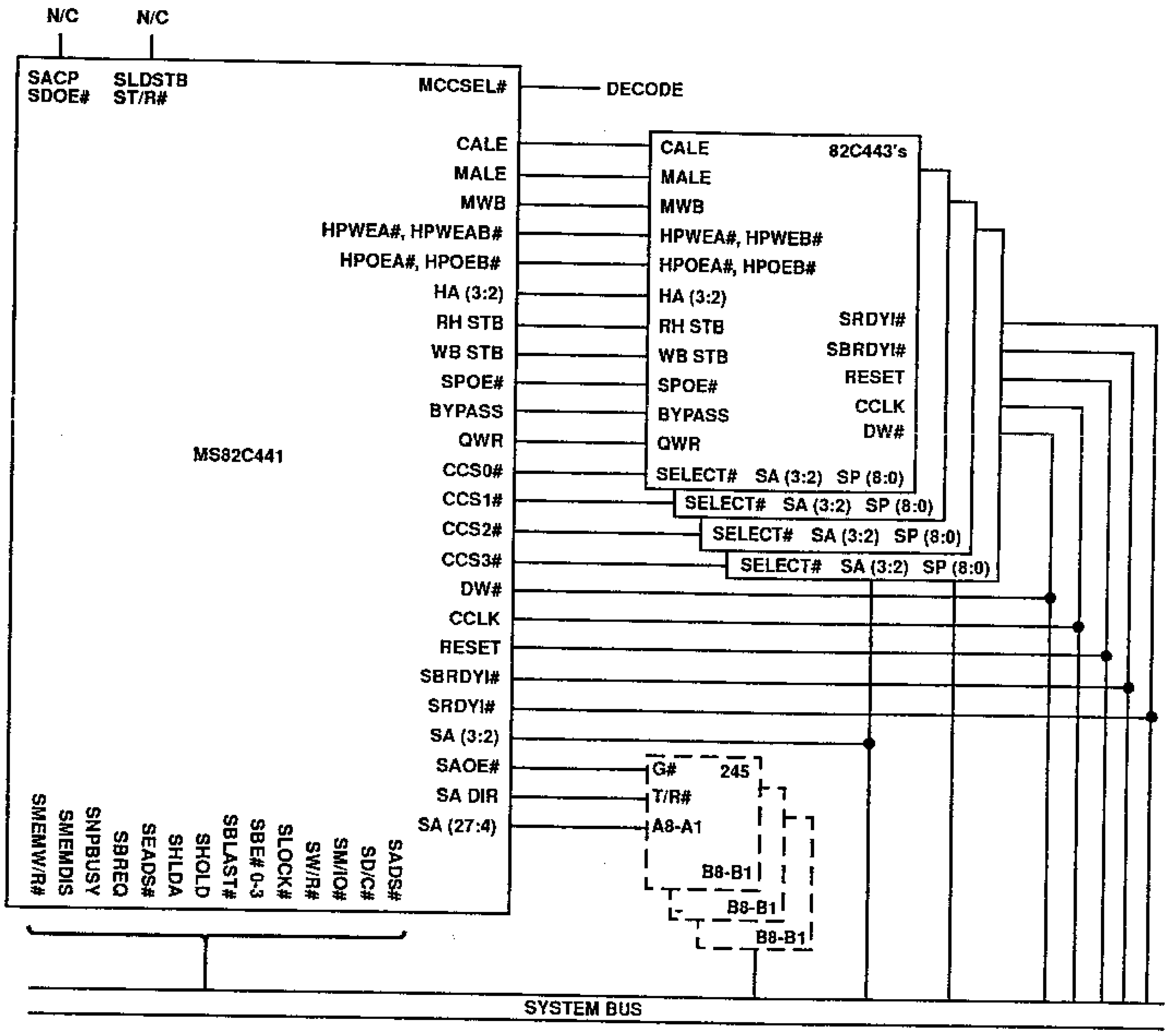
T_1 ST_i
T_2 ST_i
T_2 ST_1
T_1 ST_2
T_2 ST_2
T_2 ST_2
T_2 ST_2
T_2 ST_2
T_i ST_2
T_i ST_2
T_i ST_2
T_i ST_2
T_i ST_2
T_i ST_i
T_i ST_1
T_i ST_2
T_i ST_2
CLK
ADS #
MALE
ADDR
045
022
HA (3:2)
00
01
00
01
10
11
WBSTB
HPOEx #
HPWEx #
BYPASS
HP (8:0)

Figure 22

Section 7. MS82C441/443 Host and System Interface



MS82C441/443 Host Interface



MS82C441/443 System Interface

Section 8. DC Characteristics

DC CHARACTERISTICS (V_{CC} = 5V ±5%)

Symbol	Parameter	Test Conditions	Min.	Max.	Units
V_{IL}	Input Low Voltage	(Note 1)	–0.3	0.8	V
V_{IH}	Input High Voltage		2.2	V_{CC}+0.5	V
V_{OL}	Output Low Voltage		2.2	0.4	V
V_{OH}	Output High Voltage		2.4	—	V
I_{CC}	Power Supply	25 MHz 33 MHz	— —	200 250	mA mA
I_{LI}	Input Leakage Current	0V < V_{IN} ≤ V_{CC}	0	10	µA
I_{LO}	Output Leakage Current	0.4 < V_{OUT} < V_{CC}	0	50	µA
C_{IN}	Input Capacitance	(Note 2)	—	7	pF
C_{CLK}	CLK2 Input Capacitance	(Note 2)	—	20	pF

NOTES:
1. Minimum value is not 100% tested.
2. Sampled only.

Section 9. Package Thermal Specifications

This section gives the thermal characteristics of the 184-pin PQFP package, and the resulting maximum ambient temperature allowable to ensure proper operation of the MS82C441. Calculations were made by the formula:

$$\text{MAX } T_A = \text{MAX } T_J - P\Theta_{JA},$$

Where: MAX T_J = 105° and
$P = 5V * I_{CC\,MAX}$

	Airflow — ft/min			
	0	200	400	600
Θ_{JA}	65	60	55	50

Table 4. Θ_{JA} (°C/W) vs. Airflow — ft/min

	Airflow — ft/min			
	0	200	400	600
25 MHz Max. T_A	40	45	50	55
33 MHz Max. T_A	23	30	36	42

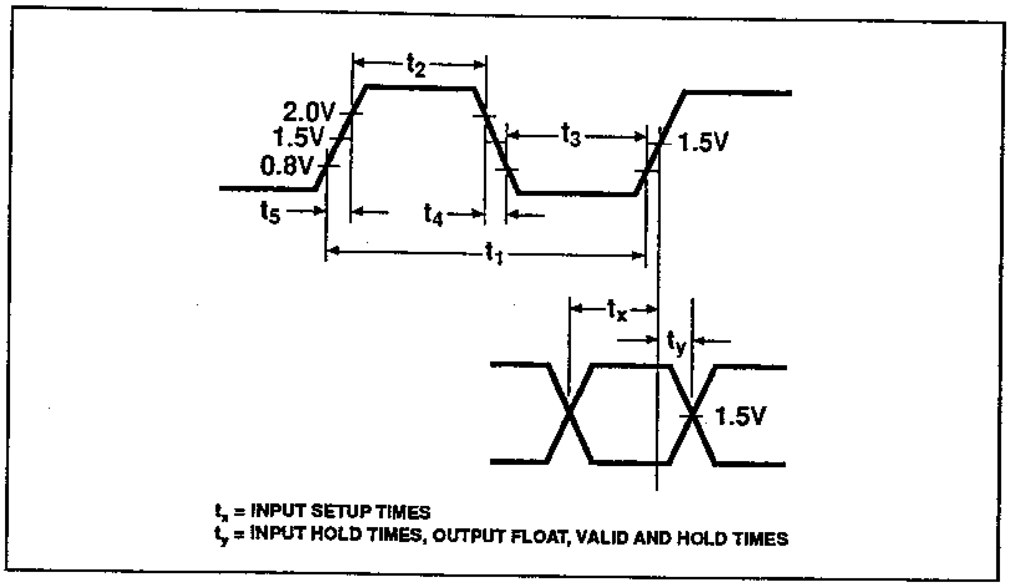
Table 5. Maximum Ambient Temperature Allowable vs. Airflow (°C)

Section 10. 25 and 33 MHz AC Characteristics

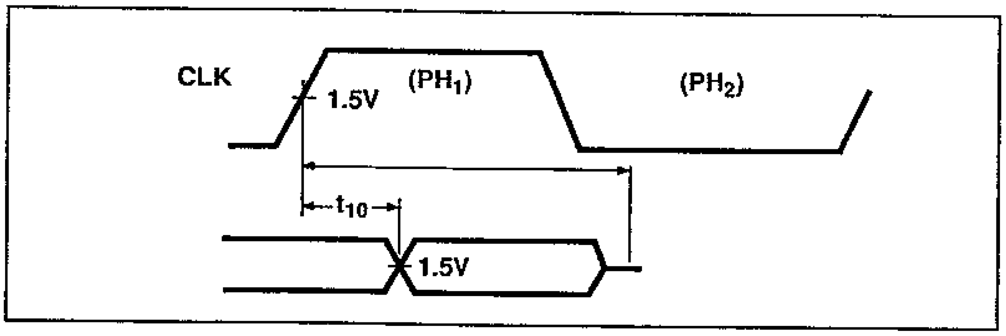
25 MHz MS82C443 AC CHARACTERISTICS — ADVANCE INFORMATION
(V_{CC} = 5V ±5%, System Address and Data C_L = 100 pF, Other C_L = 50 pF)

Symbol	Parameter	Min.	Max.	Units	Notes
	Frequency	8	33	MHz	
t_1	CLK Period	30	125	ns	
t_2	CLK High Time	11		ns	
t_3	CLK Low Time	11		ns	
t_4	CLK Fall Time		3	ns	
t_5	CLK Rise Time		3	ns	
t_6	HALE, ADS# SETUP TIME	13		ns	
t_7	HP0<8:0> READ DATA SETUP TIME	11		ns	
t_8	BMUXC<1:0> TO VALID HP DATA DELAY		?	ns	
t_9	VALID ADD TO HP VALID DATA DELAY (From Data Array)		25	ns	
t_{10}	SP<8:0> WRITE DATA VALID DELAY	3	12	ns	
t_{11}	SP<8:0> READ DATA SETUP TIME	5		ns	
t_{12}	SP<8:0> HOST READ DATA SETUP TIME	10		ns	
t_{13}	SP<8:0> READ HOLD TIME	3		ns	
t_{14}	SP TO HP BYPASS PROP DELAY		5	ns	
t_{15}	G0#, G1#, MALE, SA<3:2>, QWRWQ, SW#, MPINC, BMUXC<1:0>, RESET, SRDY#, SBRDY# SETUP TIME	5		ns	
t_{16}	SELECT#, HW0#, HW1#, HA<3:2>, BYPASS, QWR SETUP TIME	8		ns	
t_{17}	G0#, G1#, MALE, SA<3:2>, QWRWQ, SW#, MPINC, BMUXC<1:0>, RESET, SRDY#, SBRDY#, SELECT#, HW0#, HW1#, HA<3:2>, BYPASS, QWR HOLD TIME	3		ns	
t_{18}	G0#, G1# TO VALID HP DATA DELAY		8	ns	
t_{19}	HA<3:2> TO HP VALID DATA DELAY		8	ns	
t_{20}	SELECT#, BYPASS, SW# TO HP OR SP VALID DATA DELAY		8	ns	
t_{21}	SA<3:2> TO SP VALID DATA DELAY		8	ns	
t_{22}	G0#, G1#, SELECT#, SW# TO SP OR HP FLOAT DELAY *		20	ns	

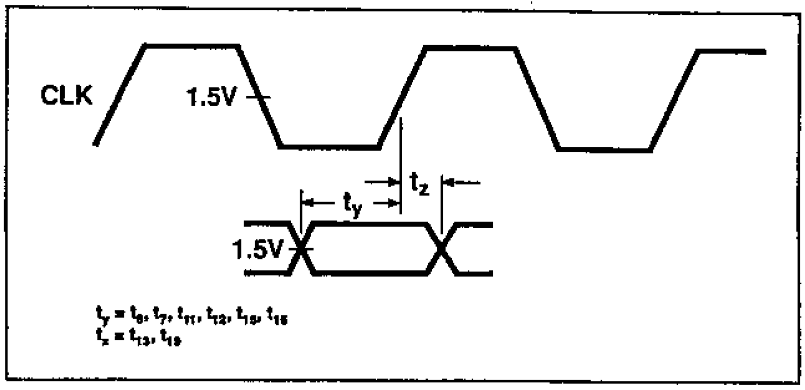
NOTES:
1. Guaranteed by characterization. Not 100% tested.
2. To guarantee recognition on a specific clock edge.


t2
2.0V
1.5V
0.8V
t3
1.5V
t5
t4
t1
tx
ty
1.5V
t_x = INPUT SETUP TIMES
t_y = INPUT HOLD TIMES, OUTPUT FLOAT, VALID AND HOLD TIMES

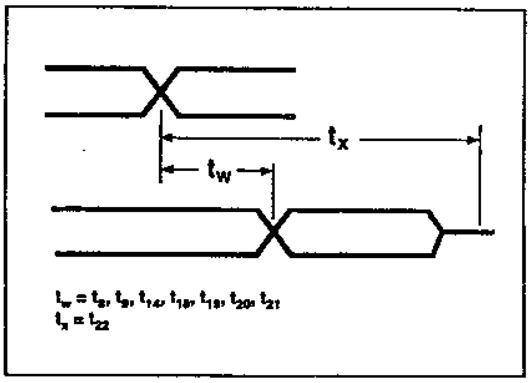
CLK Waveforms



Synchronous Output Waveforms

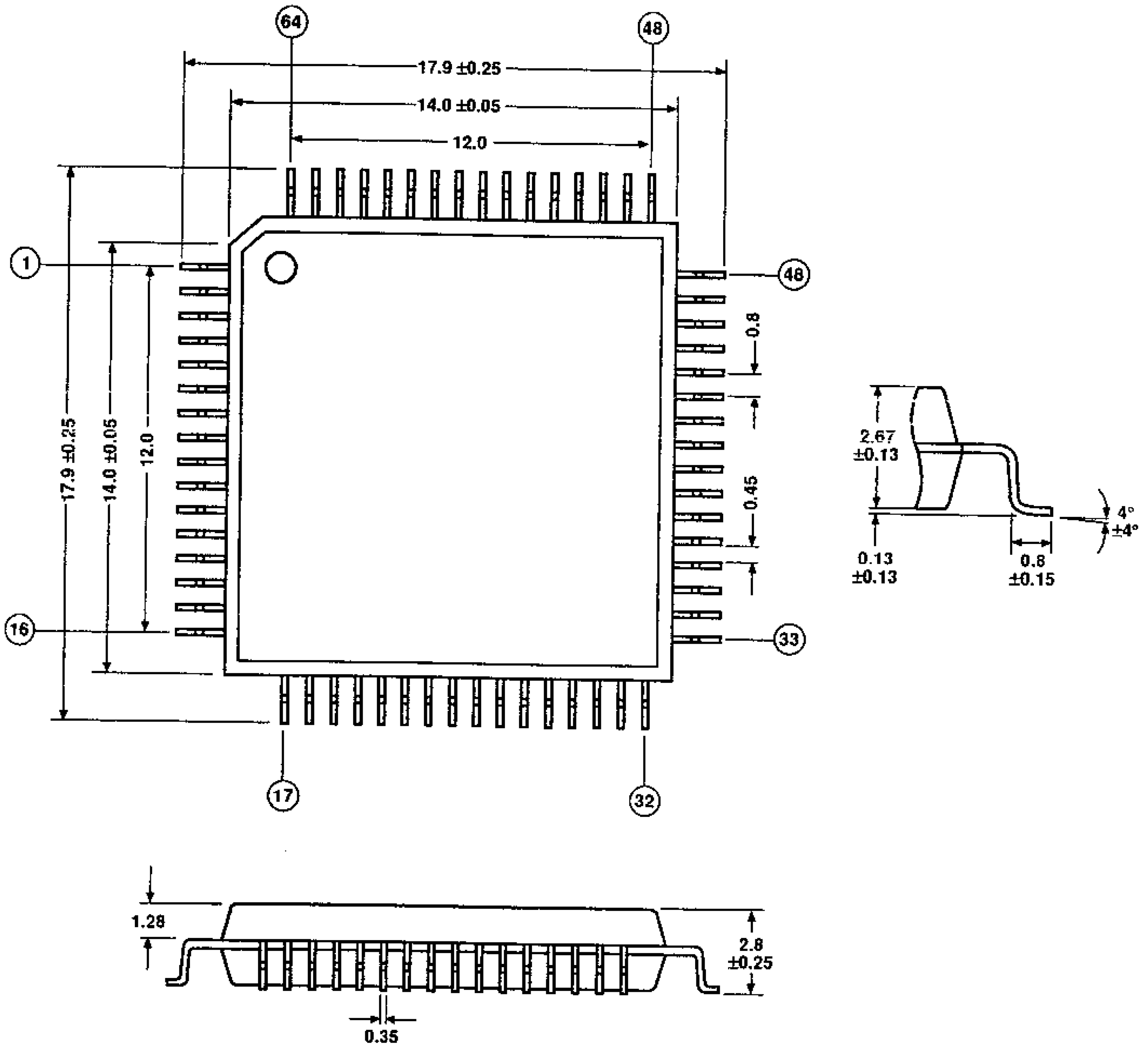


Input Waveforms



Asynchronous Output Waveforms

Section 11. Package Diagram

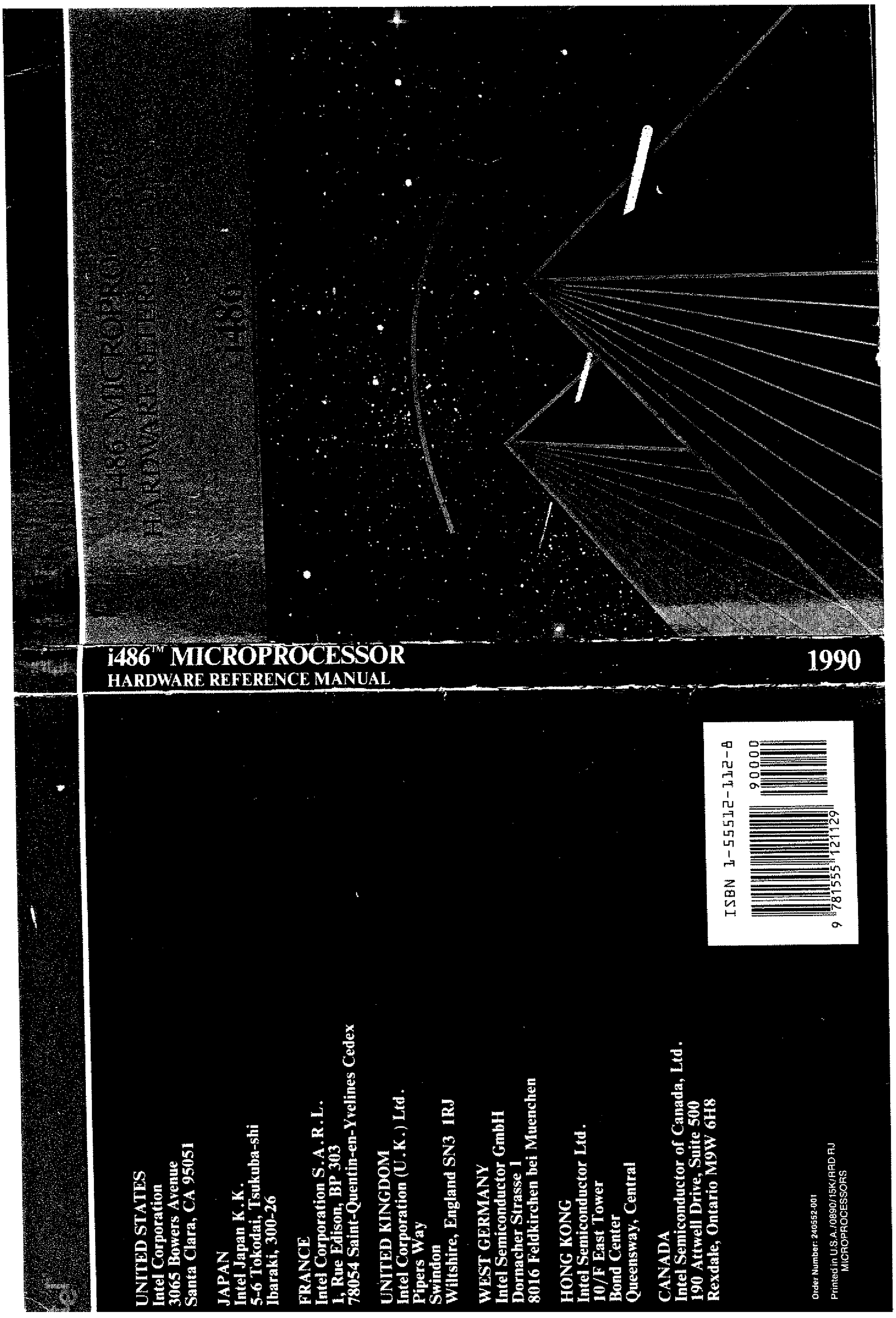


DIMENSIONS: MILIMETERS

This data sheet contains preliminary data information. MOSEL reserves the right to make changes to its products at any time without notice in order to improve design and supply the best possible product. MOSEL makes no warranty for the use of its products and bears no responsibility for any errors which may appear in this document.

SimulCache is a trademark of MOSEL Corporation.
80486 is a trademark of Intel Corporation.

EXHIBIT 3



i486™ MICROPROCESSOR
HARDWARE REFERENCE MANUAL

1990

ISBN 1-55512-112-8
90000
9 781555 121129

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison, BP 303
78054 Saint-Quentin-en-Yvelines Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Order Number: 240552-001
Printed in U.S.A./0890/15K/RRD RJ
MICROPROCESSORS

intel

INTERNATIONAL LITERATURE ORDER FORM

NAME: ______

COMPANY: ______

ADDRESS: ______

CITY: ______ STATE: ______ ZIP: ______

COUNTRY: ______

PHONE NO.: ( )

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
			X		=	
				Subtotal		
				Must Add Your Local Sales Tax		
				Total		

PAYMENT

Cheques should be made payable to your local Intel Sales Office (see inside back cover.)

Other forms of payment may be available in your country. Please contact the Literature Coordinator at your local Intel Sales Office for details.

The completed form should be marked to the attention of the LITERATURE COORDINATOR and returned to your local Intel Sales Office.

intel®

i486™ PROCESSOR HARDWARE REFERENCE MANUAL

1990

1 Introduction to the Processor

CHAPTER 1 INTRODUCTION TO THE PROCESSOR

The Intel i486™ processor is the highest-performance member of the Intel 386™ family of processors. The i486 processor executes DOS, Windows, OS/2 operating system, UNIX System V/386, iRMX* operating system, and iRMX kernel applications faster than any other processor. It is upward binary compatible with the 8086, 8088, 80186, 80286, 386 DX processor, and 386 SX processors. The i486 processor brings mainframe power to PC architectures.

1.1 ARCHITECTURE

The i486 processor includes an integer processing unit, floating-point processing unit, memory-management unit, and cache. With these units together on a single chip, many inter-unit signals remain on-chip, running at the speed of VLSI silicon rather than the speed of printed circuit boards. The increased level of integration also reduces board space, which lowers cost and simplifies design.

The i486 processor can give a two- to four-fold performance improvement over the 386 processor, depending on the clock speeds used and the specific application. Like the 386, the i486 processor includes both segment-based and page-based memory protection schemes. Instruction processing time is reduced by on-chip instruction pipelining. By performing fast, on-chip memory management and caching, the i486 processor relaxes requirements for memory response for a given level of system performance.

The i486 processor bus is significantly faster than the 386 processor (local) bus. Both buses are 32 bits wide, but the i486 processor bus introduces the use of a single-frequency (1x) clock and support for parity checking, burst cycles, cacheable cycles, cache invalidation cycles, and 8-bit data buses. There are two major advantages to using a 1x clock. First, it simplifies system design by cutting in half the clock frequency required by external devices. Second, elimination of the 2x clock used on the 386 processor reduces RF emission at the higher speed of the i486 processor and simplifies clock generation.

The i486 processor can use burst cycles for read transfers which require multiple bus cycles. Burst cycles are done at the continuous rate of one 32-bit (doubleword) transfer per clock cycle. In the 386 processor, by comparison, data transfers require at least two clock cycles per transfer. External cache, interleaved memory banks, or DRAMs with static-column addressing may be used to achieve zero wait-state memory performance during a burst.

Instructions can be executed in fewer clock cycles than with the 386 processor. In the i486 processor, streamlined instruction pipelining supports a continuous execution rate of one clock cycle per instruction for most instructions. The internal cache supports a continuous rate of one processor request per clock cycle. To support efficient task switching in real-time multitasking and multiuser systems, the i486 processor, like the 386 processor, allows a single instruction or an interrupt to perform a complete task switch.

Device testing is supported by a built-in self-test. Results of the built-in self-test are available in an internal register. Assembly-language testing of the cache and translation lookaside buffer are also supported.

Chapter 2 describes the processor's internal architecture. Chapter 3 describes the processor bus. The rest of this section highlights features of particular interest to system designers.

1.1.1 Features

The i486 processor offers the following features:

- *Compatibility*—The processor is binary-compatible with the 8086, 8088, 80186, 80286, 386 processor, and 386 SX processor.
- *Full 32-bit integer processor*—The processor performs a complete set of arithmetic and logical operations on 8-, 16-, and 32-bit data types using a full-width ALU and eight general-purpose registers.
- *Separate 32-bit Address and Data Paths*—Four gigabytes of physical memory can be addressed directly.
- *Single-Cycle Execution*—Many instructions execute in a single clock cycle.
- *On-Chip Floating-Point Unit*—The 32-, 64-, and 80-bit formats specified in IEEE Standard 754 are supported. The unit is binary-compatible with the 8087, 80287, 387™ DX coprocessor, and 387 SX coprocessor.
- *On-Chip Memory Management Unit*—Address-management and memory-space protection mechanisms maintain the integrity of memory. This is necessary in multitasking and virtual-memory environments, like those implemented by the UNIX and OS/2 operating systems. Both memory segmentation and paging are supported.
- *On-Chip Cache, with Cache Consistency Support*—The internal write-through cache can hold 8K bytes of data or instructions. Cache hits are as fast as read accesses to a processor register. Bus activity is tracked to detect alterations in the memory which internal cache represents. The internal cache can be invalidated or flushed, so that an external cache controller can maintain cache consistency in multi-processor environments.
- *External Cache Control*—Write-back and flush controls over an external cache are provided so that the processor can maintain cache consistency in multi-processor environments.
- *Instruction Pipelining*—The fetching, decoding, execution, and address translation of instructions is overlapped within the i486 processor. This results in a continuous execution rate of one clock cycle per instruction, for most instructions.

- *Burst Cycles*—Burst transfers allow a new doubleword to be read from memory each clock cycle. With this capability the internal cache and instruction prefetch buffer can be filled very rapidly.
- *Write Buffers*—The processor can continue operations internally after a write, without waiting for the write to be executed on the processor bus.
- *Bus Backoff*—If another bus master needs control of the bus during a i486 processor-bus cycle, the i486 processor will float its bus signals, then restart its cycle when the bus again becomes available.
- *Instruction Restart*—Programs can continue execution following an exception generated by an unsuccessful attempt to access memory. This feature is important for supporting demand-paged virtual memory applications.
- *Dynamic Bus Sizing*—External controllers can dynamically alter the effective width of the data bus. Bus widths of 8, 16, or 32 bits can be used.

1.1.2 Operating Modes and Compatibility

The i486 processor can run programs in modes which give it object-code compatibility with software written for the 8086, 80286, and 386 processor families. The operating mode is set in software as:

- *Real Mode:* When the processor is reset or powered up, it is initialized in Real Mode. This mode has the same base architecture as the 8086 processor but allows access to the 32-bit register set of the i486 processor. The address mechanism, maximum memory size (1 Mbyte), and interrupt handling are identical to the Real Mode of the 80286 processor. Nearly all of the i486 processor instructions are available, but the default operand size is 16 bits; in order to use the 32-bit registers and addressing modes, override instruction prefixes must be used. The primary purpose of Real Mode is to set up the processor for Protected Mode operation.
- *Protected Mode (also called Protected Virtual Address Mode):* The complete capabilities of the i486 processor become available when programs are run in the Protected Mode. In addition to segmentation protection, paging can optionally be used in Protected Mode. Linear address space is four gigabytes and virtual memory programs of up to 64 terabytes can be run. All existing 8086, 80286, and 386 processor software can be run under the i486 processor's hardware-assisted protection mechanism. The addressing mechanism is more sophisticated in Protected Mode than in Real Mode.

  Virtual 8086 Mode, a sub-mode of Protected Mode, allows 8086 programs to be run with the segmentation and paging protection mechanisms of Protected Mode. This mode offers more flexibility than the Real Mode for running 8086 programs. Using this mode, the i486 processor can execute 8086 operating systems and applications simultaneously with an i486 operating system and both 80286 and i486 processor applications.

The hardware offers additional modes which are described in Chapter 2 of this manual. For more information on operating modes, see the *i486™ Microprocessor Data Sheet* and the *i486™ Programmer's Reference Manual*.

1.1.3 Memory Management

The memory management unit supports both segmentation and paging. Segmentation provides several independent, protected address spaces. This is a security feature which limits the damage a program error can cause. For example, a program's stack space should be prevented from growing into its code space. The segmentation unit maps the separate address spaces seen by programmers into one unsegmented, linear address space.

Paging provides access to data structures larger than the available memory space by keeping them partly in memory and partly on disk. Paging breaks the linear address space into units of 4K bytes called *pages*. When a program makes its first reference to a page, the program can be stopped, the new page copied from disk, and the program restarted. Programs tend to use only a few pages at a time, so a processor with paging can simulate a large address space in RAM using a small amount of RAM, plus storage on a disk.

1.1.4 On-Chip Cache

A software-transparent 8K-byte cache stores recently accessed information on the processor chip. Both instructions and data can be cached. If the processor needs to read data which is available in the cache, the cache responds and a time-consuming external memory cycle is avoided. This allows the processor to complete transfers faster and reduces traffic on the processor bus.

The cache uses a write-through protocol; all writes to the cache are immediately passed on to the external memory which the cache represents, rather than stored for future memory updating (write-back). To reduce the impact of writes on performance, the processor can buffer write cycles; an operation which writes data to memory can finish before the write cycle is actually performed on the processor bus.

The processor performs a cache line fill to place new information into the on-chip cache. This operation reads four doublewords into a cache line, the smallest unit of storage which can be allocated in the cache. Most read cycles on the processor bus result from cache misses, which cause cache line fills.

Mechanisms are provided to maintain cache consistency between memory and cached data in multiple bus master environments. The mechanisms protect the i486 processor from reading invalid data from its own internal cache or from external caches. For example, when the i486 processor attempts to read an operand from memory that is also held in the cache of another bus master, the other bus master must be forced to write its cached data back to memory before the i486 processor can complete its read from memory. This is done because the cached version of the data may have been updated, and so may now be different from the version stored in memory.

Most memory systems optimize the speed of access on a read cycle. This is because the large majority of all memory accesses in a typical system are read accesses. The i486 processor's internal cache changes this ratio. Most read requests will result in cache hits, so most memory accesses on the processor bus will be write cycles. Memory optimization should be done with this in mind.

1.1.5 Floating-Point Unit

The internal floating-point unit performs floating-point operations on the 32-, 64- and 80-bit arithmetic formats specified in IEEE Standard 754. Like the integer processing unit, the floating-point unit architecture is binary-compatible with the 8087, 80287 coprocessors. The architecture is 100% compatible with the 387 DX coprocessor, and 387 SX coprocessor.

Floating-point instructions are executed fastest when they are entirely internal to the processor. This occurs when all operands are in the internal registers or cache. When data needs to be read from or written to external locations, burst transfers minimize the time required and a bus locking mechanism ensures that the bus is not relinquished to other bus masters during the transfer. Bus signals are provided to monitor errors in floating-point operations and to control the processor's response to such errors.

1.2 SYSTEM COMPONENTS

Intel offers several chips which are highly compatible with the i486 processor. These components can be used to design high-performance systems with a minimum of effort and cost. For components not directly connectable to the i486 processor bus, industry-standard interfaces can be used, such as the MULTIBUS II system bus.

For Ethernet interfacing, the 82596 32-bit LAN coprocessor off-loads network data management and physical-layer LAN functions to a single chip. The 82320-family 32-bit MCA system peripherals provide efficient, low-cost interfacing to Micro Channel expansion buses for PS/2 systems. The 82350-family 32-bit EISA system peripherals provide efficient, low-cost interfacing to EISA expansion buses. Several other components are currently in development.

Table 1-1 lists the components which interact directly with the i486 processor bus. Chapter 9 gives more details on many of these system peripherals. Chapter 10 describes MULTIBUS II system bus interfacing.

1.2.1 i486 Processor

The i486 processor provides all of the integer and floating-point CPU functions plus many of the peripheral functions required in a typical computer system. It executes the complete instruction set of the 386 processor and 387 DX numerics coprocessor, with some extensions. The processor eliminates the need for an external memory management unit, and the on-chip cache minimizes the need for external cache and associated control logic.

Table 1-1. System Components

Component	Name	Description
32-Bit General-Purpose CPU	i486™ CPU	General-purpose processor with floating-point arithmetic, memory management, and cache.
32-Bit LAN Coprocessor	82596CA	Local-area network communications coprocessor supporting CSMA/CD protocol.
32-Bit MCA System Peripherals	82320	Functional support for Micro Channel (PS/2) expansion buses and boards. Seven chips in the set.
32-Bit EISA System Peripherals	82350	Functional support for EISA expansion buses and boards. Four chips in the set.
485Turbocache Module for i486 microprocessor	485Turbocache Module	Second-level cache module.

Chapters 2 through 7 of this manual focus on the details of the i486 processor's architecture, hardware functions, and interfacing. For more information on the architecture and software interface, see the *i486™ Processor Programmer's Reference Manual.*

1.2.2 LAN Coprocessor

The 82596CA LAN coprocessor is a 32-bit multitasking local-area network communications processor that supports 80-Mbyte/second transfers at 25 MHz. It implements the carrier-sense, multiple-access and collision-detect (CSMA/CD) link access protocol and interfaces the i486 processor to a wide variety of networks and functions, including:

- IEEE 802.3 networks (Ethernet, HDLC, Cheapernet, StarLAN, and others).
- IBM PC networks (baseband and broadband).
- Proprietary CSMA/CD networks.
- HDLC frame delimiting.

The 82596 LAN coprocessor is typically used in desktop computers, file servers, and gateways. It provides a high-performance front-end controller for heavy data traffic, and it permits extensive protocol-layer software implementations. A complete hardware interface to Ethernet networks, for example, can be implemented with the 82596 LAN coprocessor and the 82C501AD Ethernet serial interface device. The i486 processor and 82596 LAN coprocessor communicate by means of a memory-based mailbox, command system and buffer system. The coprocessor fetches and executes high-level commands from shared memory to control all time-critical network functions. It performs command chaining and inter-processor communication. It is object-code compatible with the 82586 LAN coprocessor, with extensions which simplify software drivers.

Because the 82596 LAN coprocessor can execute commands directly from main memory and operate on data buffers without processor intervention, supervision from the i486 processor is minimized. In large networks, the high performance of the i486 processor in executing control and protocol software minimizes the need for host intervention.

1.2.3 485Turbocache Module

The 485Turbocache Module is a high-performance, optional, write-through, second-level cache designed specifically for the i486 microprocessor. It consists of the 82485 cache controller and 4 to 8 custom SRAMs for a complete cache solution in one package.

The 485Turbocache Module is a performance upgrade for 25-MHz or 33-MHz i486 microprocessor systems. One module provides 64K or 128K bytes of external cache memory. Up to four modules may be cascaded for up to 512K bytes of external cache memory. The module is optional, that is a single socket allows three price/performance options: no cache, a 64K cache, or a 128K cache.

The module is organized as two-way, set-associative with a line size of 16 bytes. The interface to the i486 microprocessor is simple since all CPU timings and bus cycles are supported. The module also supports Burst Mode, BOFF# cycles, and the same invalidation cycles as the processor.

While performance benefits are extremely application sensitive, the module typically provides from 5% to 30% performance improvement. The 485Turbocache Module provides the best price-performance ratio for 25- and 33-MHz i486 microprocessor designs. Chapter 6 discusses the 485Turbocache Module in detail.

1.2.4 EISA Chip Set

The 82350 family of peripherals interfaces the i486 processor to an extended industry standard architecture (EISA) bus. The chip set includes three motherboard peripherals (bus controller, integrated system peripheral, and bus buffers) and one peripheral for EISA-bus expansion boards (a bus master interface chip). The EISA standard maintains full compatibility with the existing ISA (also known as AT) standard. The EISA expansion board connector is a superset of the ISA expansion board connector, allowing existing 8- and 16-bit ISA boards to be installed in EISA slots. This is discussed in detail in Chapter 8.

The EISA bus controller performs data path translation, bus timing, and centralized bus arbitration. The improvements over the ISA standard are provided transparently, even to existing ISA DMA devices.

The EISA integrated system peripheral contains most of the EISA-specific peripheral functions, including DMA controller, 2 eight-channel interrupt controllers, 4 counter modules, EISA bus arbiter, and DRAM refresh address generator. The peripheral operates in a tightly coupled environment with the EISA bus controller to generate control signals for the DMA transfers. A master on any of the buses can communicate in parallel with both devices. Transfers between buses of varying sizes or transfers with misaligned addresses are performed correctly.

The EISA bus-master interface controller is the primary interface between local functions on an EISA expansion board and the EISA bus on the i486 system motherboard. The primary function of the controller is to support burst transfers between the expansion board and main memory. Data transfer rates of up to 33 Mbytes/second are supported – the fastest available on an EISA bus. With the controller, an EISA expansion board can be implemented with simple logic similar to that used in traditional ISA DMA designs. The general-purpose command and status interface allows a variety of software control protocols by a local expansion-board processor. Data transfers on the local processor bus are similar to traditional DMA transfer protocols. Local processors are supported with the ability to access individual locations in system memory or I/O space.

1.2.5 High-Performance PLDs

Programmable Logic Devices (PLDs) have become a vital factor in systems design. Intel manufactures a line of CMOS PLDs that meet the performance requirements of high-speed systems while reducing power consumption and heat dissipation. Some of these devices, such as the 85C220 (20-pin general-purpose PLD), 85C224 (24-pin general-purpose PLD), and 85C508 (28-pin address decoder PLD), are shown in this manual.

The 85C220 and 85C224 PLDs are both supersets to commonly used bipolar and CMOS alternatives (16x8 and 20x8 type devices). Both Intel PLDs are available at clock speeds to support fast state-machines in i486 systems. The 85C508 is a 28-pin address decoder PLD with integral transparent latches on its eight outputs.

1.3 SYSTEM ARCHITECTURE

The i486 processor can be the foundation for systems ranging from single-processor to multiprocessor. A single-processor system might be a personal computer, updated to use the i486 processor. A system design of this type offers higher performance through the integration of floating-point processing, memory management, and caching. More complex systems may use multiple processors which provide, at chip-level, the equivalent of board-level functions. Designs of this type are typically used in multiuser machines, scientific workstations, and engineering workstations.

A typical system, something between a single-processor design and a more complex multiprocessing design, is shown in Figure 1-1. This example uses a single i486 processor with external cache and the 82596 LAN coprocessor. Other examples of system design are illustrated in the figures that follow.

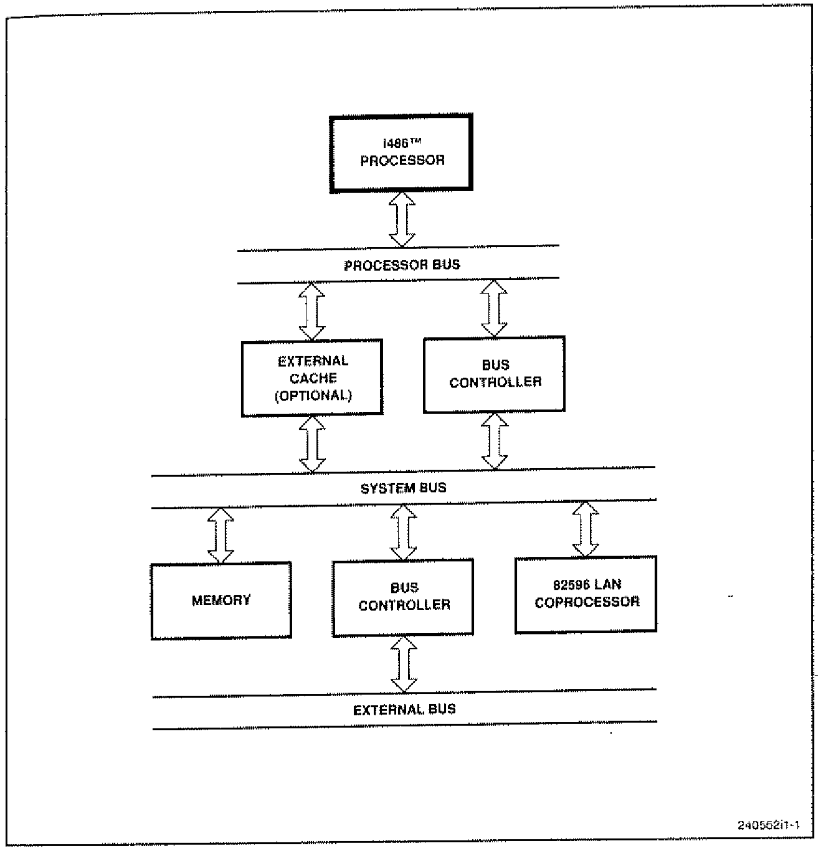


Figure 1-1. A Typical i486™ Processor System

1.3.1 Single Processor System

In single-processor systems, the processor handles all peripheral resources and intelligent devices, and executes all software. The i486 processor does this in a more efficient way and for a wider range of task complexity than earlier processors. Single-processor systems offer small size and low cost in exchange for flexibility in upgrading or expanding the system. Typical applications include personal computers, small desktop workstations and embedded controllers. Such applications are implemented as a single board, usually called a motherboard; the processor bus does not extend beyond the board occupied by the i486 processor.

Figure 1-2 shows an example of such a system. In a single-processor system, devices which share the processor bus must be selected carefully. All components must interact directly with the processor bus or have interface logic which allows them to do so. The total bus bandwidth requirements of other components should be no more than 50% of the available processor-bus bandwidth. Traffic above 50% will degrade performance of the processor.

Two basic design approaches are used to elaborate the single-processor system into more complex systems. The first approach is to add more devices to the processor bus. This can be done up to the limit mentioned above: no more than 50% of the processor-bus bandwidth should be used by devices other than the i486 processor. The second design approach is to add more buses to the system. By adding buses, greater bus bandwidth is created in the system as a whole, which in turn allows more devices to be added to the system. The two approaches go hand-in-hand to expand the capabilities of a system. The sections below give only a few examples of the great design variety that is possible with devices that operate compatibly with the i486 processor.

1.3.2 Loosely Coupled Multiprocessor System

Loosely coupled multiprocessor systems include board-level products which communicate with one another through a standard system bus, such as the MULTIBUS II system bus described in Chapter 9. In this architecture, each board contains a processor and

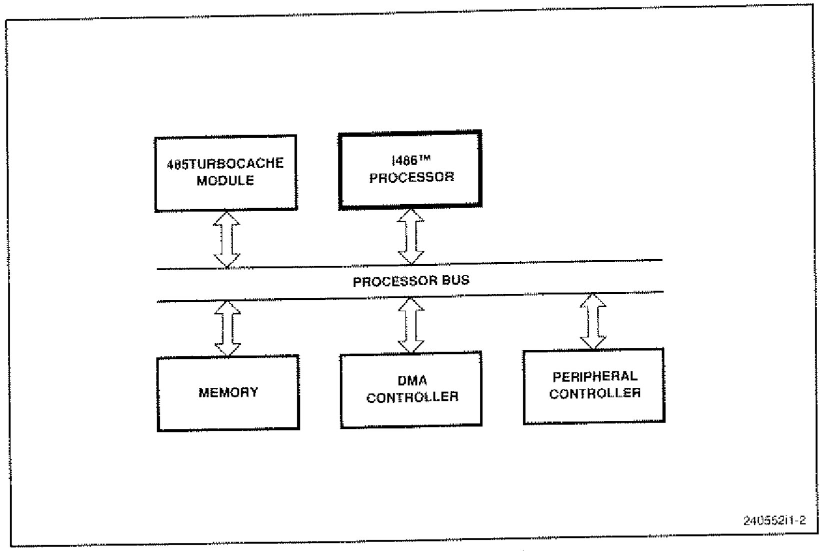


Figure 1-2. Single-Processor System

associated logic. There is typically only one processor per board. Components within each board communicate on either a processor bus or on the buffered system bus. The system bus provides extra bandwidth beyond the processor bus.

A typical system is shown in Figure 1-3. Such system-bus boards typically occur in higher-end personal computers and systems which allow for modular expansion. A typical design would include a coprocessor or LAN interface board in a personal computer, or a network-interface board in a file server or gateway. Systems built from these boards can contain a mix of processor types. Devices attached to the processor bus on a given board make demands which may affect system performance. For example, the 82596 LAN coprocessor may use up to 3% of the bus bandwidth to handle 10-Mbit/second Ethernet traffic.

1.3.3 External Cache

External cache allows a system to achieve maximum performance. This cache is essential in tightly coupled multi-processor systems. The external cache should consist of cache memory (usually fast SRAM) and cache control logic.

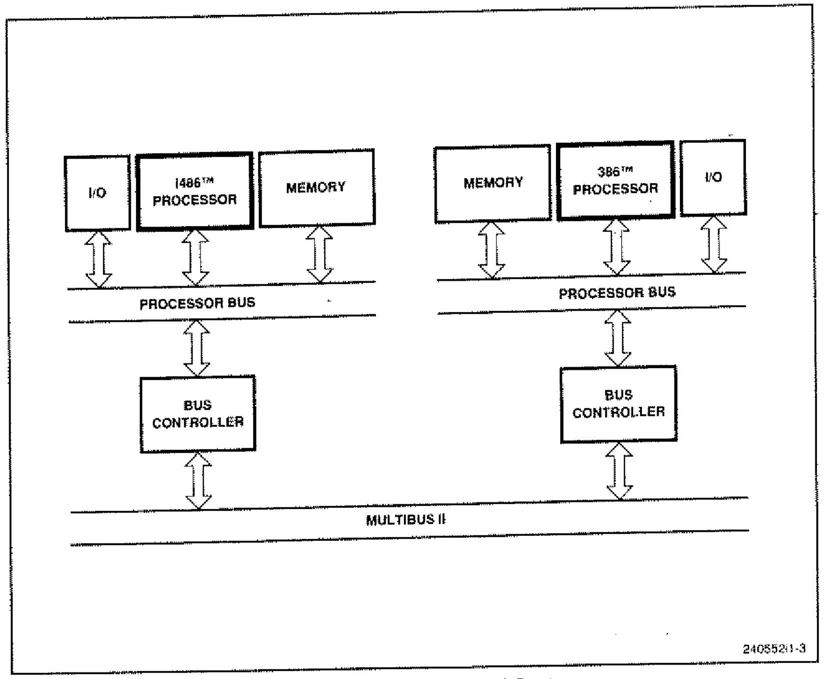


Figure 1-3. Loosely Coupled System

External cache systems typically provide access to the cache from both the processor and the system buses. This is shown in Figure 1-4. These caches typically monitor processor memory accesses, optimal mix of data and instructions, processor access time, and consistency between cache and memory.

1.4 System Applications

A majority of i486 processor systems can be grouped into these types:

- Personal Computers
- Minicomputers and Workstations
- Embedded Controllers

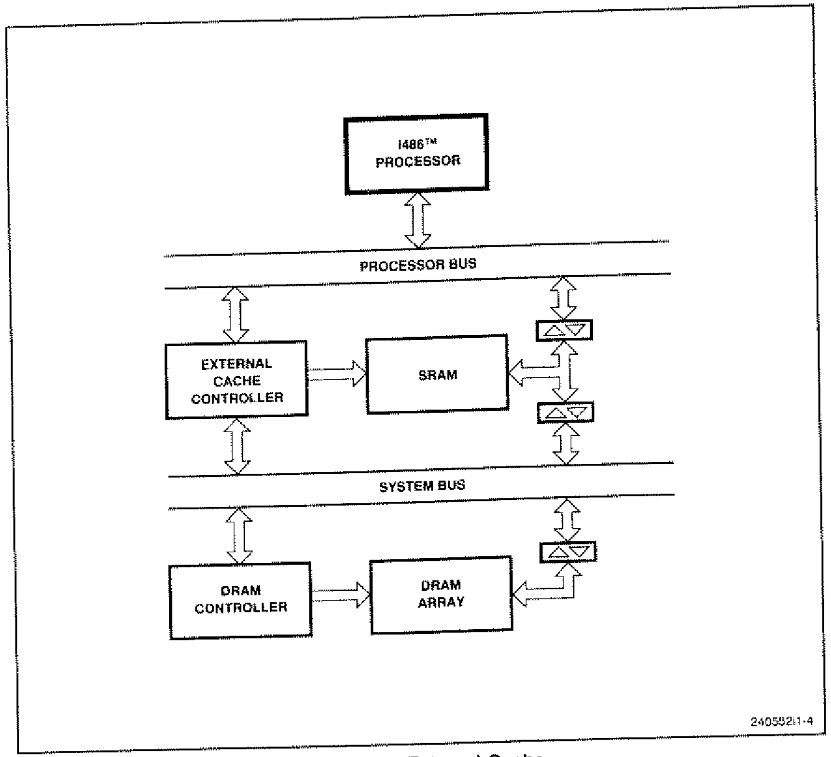


Figure 1-4. External Cache

Each type of system has distinct design goals and constraints, as described in the following sections. Software running on the processor, even in standalone embedded applications, should use a standard operating system such as DOS, Windows, OS/2 operating system, UNIX System V/386, iRMX operating system, or iRMX kernel for ease of debugging, documentation, and transportability.

1.4.1 Personal Computers

In single-processor systems, the processor will interact directly with I/O devices and DRAM memory. Other bus masters, such as the 82596 LAN coprocessor, typically reside on the system bus; conventional personal computer architecture puts most peripherals on separate plug-in boards. Expansion is typically limited to memory boards and I/O boards. A standard I/O architecture such as MCA or EISA is used. System cost and size are very important. Figure 1-5 shows an example of a personal computer application.

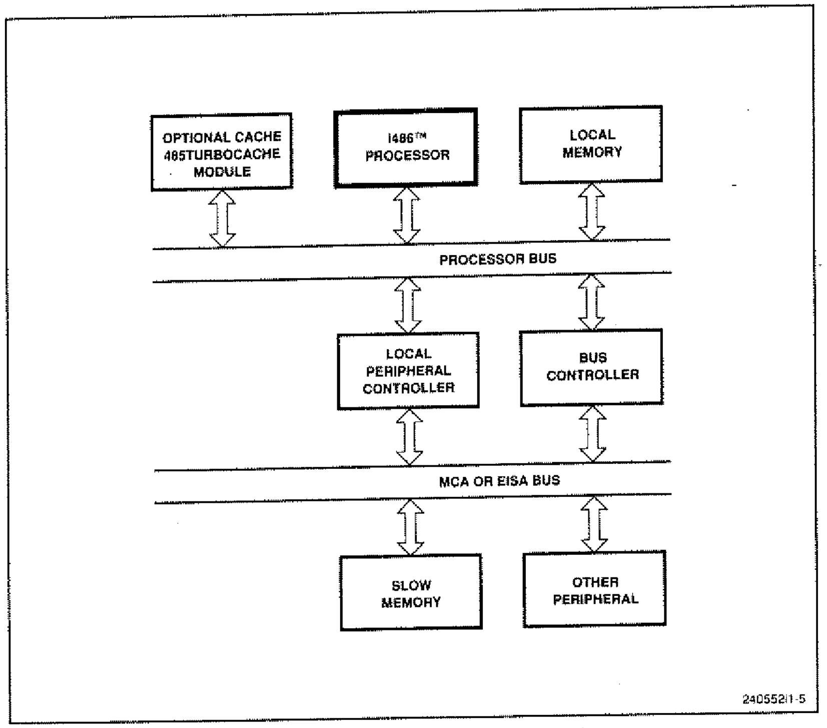


Figure 1-5. Personal Computer Example

External cache is optional in such environments, particularly if system performance is not a critical parameter. Where an external cache is used, memory-access speed will improve only if the cache is designed as a write-back system and memory access has zero to one wait states.

1.4.2 Minicomputers and Workstations

Minicomputer and workstation systems can be implemented with a loosely coupled architecture. These typically allow expansion of the number of CPUs, memory modules, and I/O devices. Standard system buses like the MULTIBUS II system bus are used. Minicomputers and workstations are more performance oriented and less cost oriented than personal computers. Higher-performance systems may need a tightly coupled architecture. Due to the variety of architectures in which minicomputers and workstations are implemented, no representative design example can be given.

The high performance of the i486 processor will cloud current distinctions between personal computers, minicomputers, and workstations. Personal computers can be viewed as lower-cost minicomputers sharing software and data with desktop workstations. Unlike personal computers, minicomputers are likely to use ECC memory and an external cache. Fast communication controllers such as the 82596 LAN coprocessor can be used on the processor bus. File servers can be designed to allow multiple communication links on the same board, connecting directly to other 82596 LAN coprocessors on the processor bus.

1.4.3 Embedded Controllers

Most embedded controllers perform real-time tasks. The performance of the i486 processor and its compatibility with the extensive 386 processor installed base are important factors in its choice. Embedded controllers are usually implemented as standalone systems, with less expansion capability than other applications because they are tailored so specifically to a single environment.

If code must be stored in EPROM or ROM for non-volatility, but performance is also a critical issue, then the code should be copied into RAM which is provided specifically for this purpose. Frequently used routines and variables, such as interrupt handlers and interrupt stacks, can be locked in the processor's internal cache so that they are always available quickly.

Embedded controllers usually require less memory than other applications, and control programs are usually tightly written machine-level routines which need optimal performance in a limited variety of tasks. The processor typically interacts directly with I/O devices and DRAM memory. Other peripherals connect to the system bus, as shown in Figure 1-6.

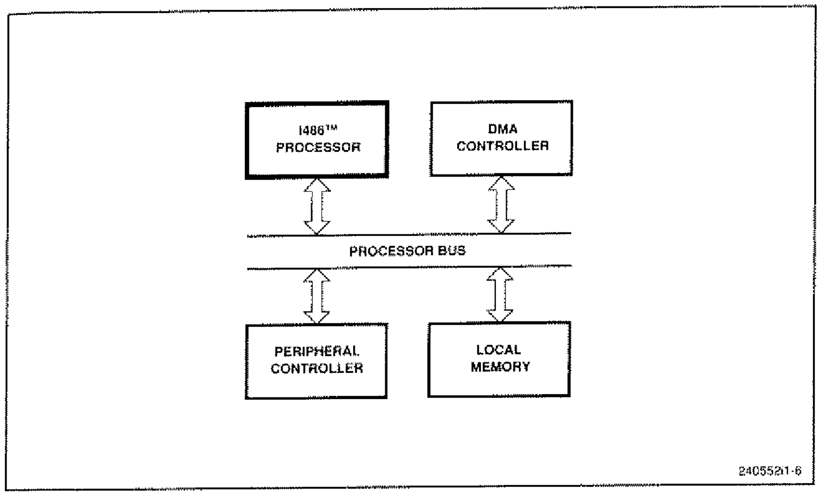


Figure 1-6. Embedded Controller Example

2 Internal Architecture

CHAPTER 2
INTERNAL ARCHITECTURE

Internally, the i486™ processor has nine functional units which operate in parallel. Figure 2-1 shows the nine internal units:

- Bus Interface
- Cache
- Instruction Prefetch
- Instruction Decode
- Control
- Integer and Datapath
- Floating-Point
- Segmentation
- Paging

The internal architecture is very much like that of the 386 processor, except for the new on-chip cache and floating-point units.

Signals from the external 32-bit processor bus reach the internal units through the bus interface unit. On the internal side, the bus interface unit and cache unit pass addresses bidirectionally through a 32-bit bidirectional bus. Data is passed from the cache to the bus interface unit on a 32-bit data bus. The closely coupled cache and instruction prefetch units simultaneously receive instruction prefetches from the bus interface unit over a shared 32-bit data bus, which is also used by the cache to receive operands and other types of data. Instructions in the cache are accessible to the instruction prefetch unit, which contains a 32-byte queue of instructions waiting to be executed.

When internal requests for data or instructions can be satisfied from the cache, time-consuming cycles on the external processor bus are avoided. The bus interface unit is only involved when an operation needs access to the processor bus. Many internal operations are therefore transparent to the external system.

The instruction decode unit translates instructions into low-level control signals and microcode entry points. The control unit executes microcode and controls the integer, floating-point, and segmentation units. Computation results are placed in internal registers within the integer or floating-point units, or in the cache. Internal storage locations (datapaths) are kept in the integer unit.

The cache shares two 32-bit data buses with the segmentation, integer, and floating-point units. These two buses can be used together as a 64-bit interunit transfer bus. When 64-bit segment descriptors are passed from the cache to the segmentation unit, 32 bits are passed directly over one data bus and the other 32 bits are passed through the integer unit, so that all 64 bits reach the segmentation unit simultaneously.

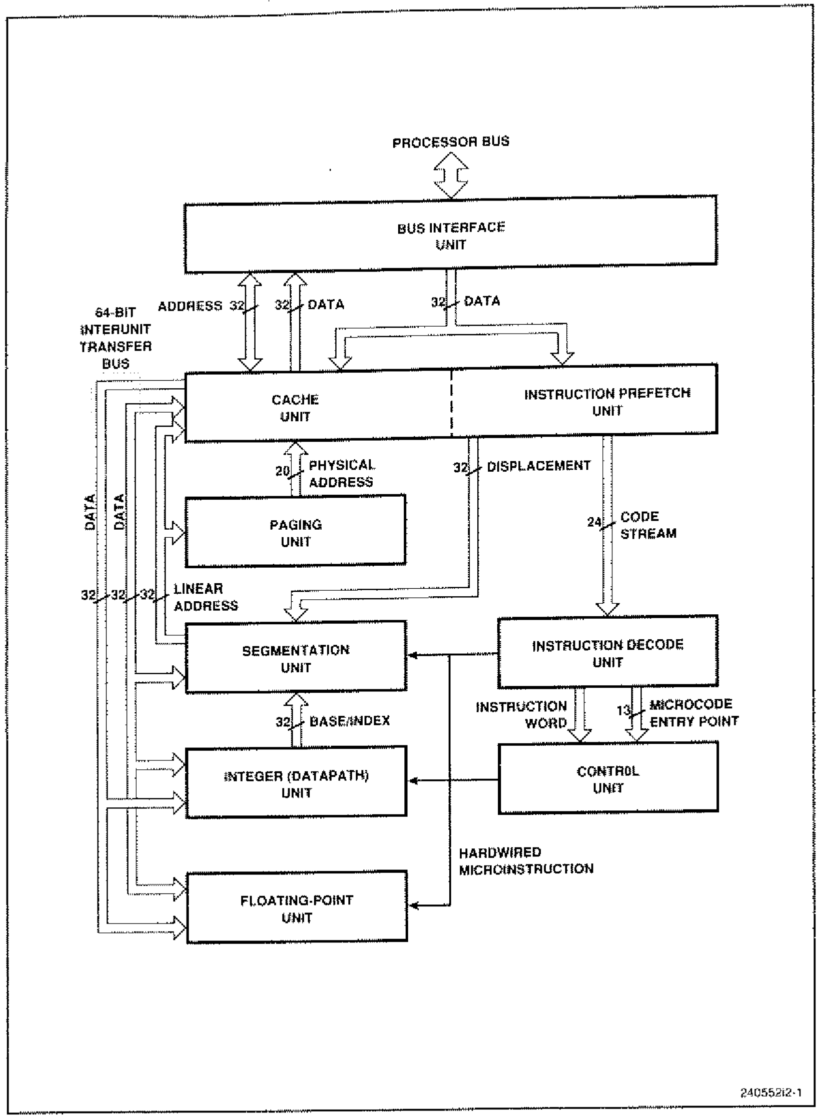


Figure 2-1. Internal Architecture

Address generation is performed by the segmentation and paging units. Logical addresses are translated by the segmentation unit and passed to the paging and cache units on a 32-bit linear address bus. The paging unit translates linear addresses into physical addresses, which are passed to the cache on a 20-bit bus.

The next section describes the internal instruction pipelining method. Following that, Sections 2.2 through 2.10 describe each of the nine internal units.

2.1 Instruction Pipelining

Not every instruction involves all internal units. When an instruction needs the participation of several units, each unit operates in parallel with others on instructions at different stages of execution. Although each instruction is processed sequentially, several instructions are at varying stages of execution in the processor at any given time. This is called *instruction pipelining*. Instruction prefetch, instruction decode, microcode execution, integer operations, floating-point operations, segmentation, paging, cache management, and bus interface operations are all performed simultaneously. Figure 2-2 shows some of this parallelism for a single instruction: the instruction fetch, 2-stage decode, execution, and register write-back of the execution result. Each stage in this pipeline can occur in one clock cycle.

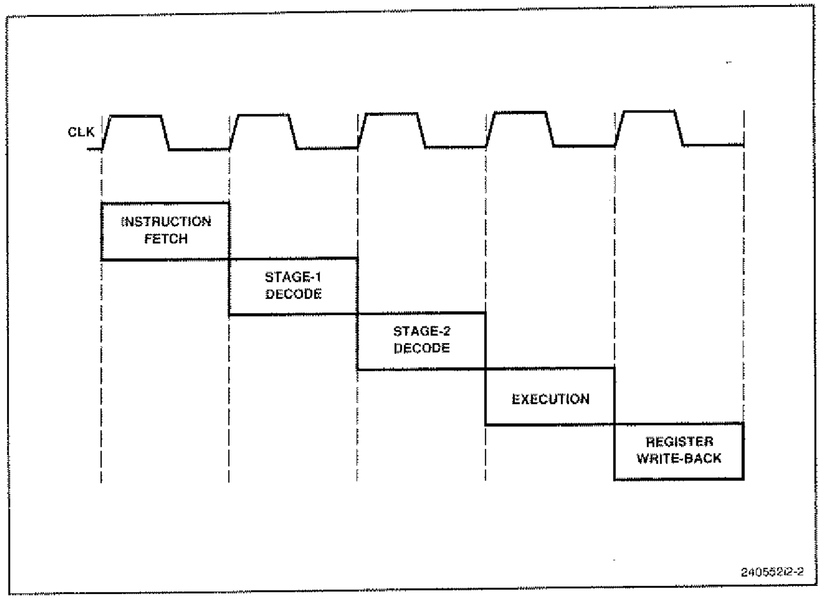


Figure 2-2. Internal Pipelining

The internal pipelining on the i486 processor offers an important performance advantage over many single-clock RISC processors: in the i486 processor, data can be loaded from the cache with one instruction and used by the next instruction in the next clock. This performance advantage results from the stage-1 decode step, which initiates memory accesses before the execution cycle. Because most compilers and application programs follow load instructions with instructions which operate on the loaded data, this method optimizes the execution of existing binary code.

The method has a performance tradeoff: an instruction sequence which changes register contents and then uses that register in the next instruction to access memory takes three clocks rather than two. This tradeoff is only a minor disadvantage, however, since most instructions which access memory use the stable contents of the stack pointer or frame pointer, and the additional clock is not used very often. Compilers often place an unrelated instruction between one which changes an addressing register and one which uses the register. Such code is compatible with the 386 processor, and the i486 processor provides special stack increment/decrement hardware and an extra register port to execute back-to-back stack push/pop instructions in a single clock.

2.2 Bus Interface Unit

The bus interface unit prioritizes and coordinates data transfers, instruction prefetches, and control functions between the processor's internal units and the outside system. Internally, the bus interface unit communicates with the cache and the instruction prefetch units through three 32-bit buses, as shown in Figure 2-1. Externally, the bus interface unit provides the processor bus signals, described in Chapter 3. Except for cycle definition signals, all external bus cycles – memory reads, instruction prefetches, cache line fills, etc. – look like conventional microprocessor cycles to external hardware, with all cycles having the same bus timing.

The bus interface unit contains the following architectural features:

- *Address Transceivers and Drivers* – The A2-A31 address signals are driven on the processor bus, together with their corresponding byte-enable signals, BE0#-BE3#. The high-order 28 address signals are bidirectional, allowing external logic to drive cache invalidation addresses into the processor.
- *Data Bus Transceivers* – The D0-D31 data signals are driven onto and received from the processor bus.
- *Bus Size Control* – Three sizes of external data bus can be used – 32, 16, and 8 bits wide. Two inputs from external logic specify the width to be used. Bus size can be changed on a cycle-by-cycle basis.
- *Write Buffering* – Up to four write requests can be buffered, allowing many internal operations to continue without waiting for write cycles to be completed on the processor bus.
- *Bus Cycles and Bus Control* – A large selection of bus cycles and control functions are supported, including burst transfers, non-burst transfers (single- and multiple-cycle), bus arbitration (bus request, bus hold, bus hold acknowledge, bus locking, bus pseudo-locking, and bus backoff), floating-point error signalling, interrupts, and reset. Two software-controlled outputs enable page caching on a cycle-by-cycle basis. One input and one output are provided for controlling burst read transfers.

- *Parity Generation and Control* – Even parity is generated on writes to the processor and checked on reads. An error signal indicates a read parity error.
- *Cache Control* – Cache control and consistency operations are supported. Three inputs allow the external system to control the consistency of data stored in the internal cache unit. Two special bus cycles allow the processor control the consistency of external cache.

2.2.1 Data Transfers

To support the cache, the bus interface unit reads 16-byte cacheable transfers of operands, instructions, and other data on the processor bus and passes them to the cache unit. When cache contents are updated from an internal source, such as a register, the bus interface unit writes the updated cache information to the external system. Non-cacheable read transfers are passed through the cache to the integer or floating-point units.

During instruction prefetch, the bus interface unit reads instructions on the processor bus and passes them to both the instruction prefetch unit and the cache. The instruction prefetch unit may then obtain its inputs directly from the cache.

2.2.2 Write Buffers

The bus interface unit has temporary storage for buffering up to four 32-bit write transfers to memory. Addresses, data, or control information can be buffered. Single I/O-mapped writes are not buffered, although multiple I/O writes may be buffered. The buffers can accept memory writes as fast as one per clock. Once a write request is buffered, the internal unit which generated the request is free to continue processing. If no higher-priority request is pending and the bus is free, the transfer is propagated as an immediate write cycle to the processor bus. When all four write buffers are full, any subsequent write transfer will stall inside the processor until a write buffer becomes available.

The bus interface unit can re-order pending reads in front of buffered writes. This is done because pending reads can prevent an internal unit from continuing, whereas buffered writes need not have a detrimental effect on processing speed.

Writes are propagated to the processor bus in the same first-in-first-out order in which they are received from the internal unit. However, a subsequently generated read request (data or instruction) may be re-ordered in front of buffered writes. As a protection against reading invalid data, this re-ordering of reads in front of buffered writes will only occur if all buffered writes are cache hits. Because an external read will only be generated for a cache miss, and will only be re-ordered in front of buffered writes if all such buffered writes are cache hits, any read generated on the external bus with this protection will never read a location which is about to be written by a buffered write.

This re-ordering can only happen once for a given set of buffered writes, because the data returned by the read cycle could otherwise replace data about to be written from the write buffers.

To ensure that no more than one such re-ordering is done for a given set of buffered writes, all buffered writes are re-flagged as cache misses when a read request is re-ordered ahead of them. Buffered writes thus marked are propagated to the processor bus before the next read request is acted upon. Invalidation of data in the internal cache also causes all pending writes to be flagged as cache misses. Disabling the cache unit disables the write buffers, which eliminates any possibility of re-ordering bus cycles.

2.2.3 Locked Cycles

The processor can generate signals to lock a contiguous series of bus cycles. These cycles can then be performed without interference from other bus masters, if external logic observes these locking signals. One example of a locked operation is a semaphor read-modify-write update, where a resource control register is updated. No other operations should be allowed on the bus until the entire locked semaphor update is completed.

When a locked read cycle is generated, the read is not attempted from the internal cache. All pending writes in the buffer are completed first. Only then is the read part of the locked operation performed, the data modified, the result placed in a write buffer, and a write cycle performed on the processor bus. This sequence of operations ensures that all writes are performed in the order in which they were generated.

2.2.4 I/O Transfers

Transfers to and from I/O locations have some restrictions to ensure data integrity:

- *Caching*—I/O reads are never cached.
- *Read Re-Ordering*—I/O reads are never re-ordered ahead of buffered writes to memory. This ensures that the processor will have completed updating all memory locations before reading status from a device.
- *Writes*—Single I/O writes are never buffered. Thus, when processing an OUT instruction, internal execution stops until all buffered writes and the I/O write are completed on the processor bus. This allows time for external logic to drive a cache invalidate cycle or mask interrupts before the processor executes the next instruction. The processor will have completed updating all memory locations before writing to the I/O location. However, repeated OUT instructions may be buffered.

I/O device recovery time; is determined by the write buffers and the cache unit. In the 386 processor, back-to-back write recovery time; could be guaranteed to exceed a certain value by inserting a jump to the next instruction that writes to the I/O device. This forced an instruction prefetch cycle which could only be performed after the preceding write was completed. This technique is not used in the i486 processor because a prefetch can be satisfied internally by the cache and recovery time may be too short. The same effect

is achieved in the i486 processor by explicitly generating a read to an area of memory that is not cacheable. Because the i486 processor does not buffer single I/O writes, such a read will not be done until the I/O write is completed.

2.3 Cache Unit

The cache unit stores copies of recently read instructions, operands, and other data. When the processor requests information already in the cache—called a *cache hit*—no processor-bus cycle is required. When the processor requests information not in the cache—called a *cache miss*—the information is read into the cache in one or more 16-byte cacheable data transfers, called *cache line fills*. When an internal write request is generated to an area currently in the cache, two things happen: the cache is updated, and the write is also passed through the cache to memory. This is called *cache write-through*.

The cache transfers data to other units on two 32-bit buses, as shown in Figure 2-1. The cache receives linear addresses on a 32-bit bus and the corresponding physical addresses on a 20-bit bus. The cache and instruction prefetch; units are closely coupled. 16-byte blocks of instructions in the cache can be passed quickly to the instruction prefetch unit. Both units read information in 16-byte blocks.

The cache can be accessed as often as once each clock. The cache acts on physical addresses, which minimizes the number of times the cache must be flushed. When both the cache and the cache write-through functions are disabled, the cache may be used as a high-speed RAM.

2.3.1 Cache Structure

The cache has a *four-way set associative* organization. There are four possible cache locations to store data from a given area of memory. Four-way association is a compromise between the speed of a direct-mapped cache during cache hits, and the high cache-hit ratio of fully associative cache. As shown in Figure 2-3, the 8-Kbyte data block is divided into four data *ways*, each containing 128 16-byte *sets*, or *cache lines*. Each cache line holds data from 16 successive byte addresses in memory, beginning with an address divisible by 16.

Cache addressing is performed by dividing the high-order 28 bits of the physical address into three parts, as shown in Figure 2-3. The 7 bits of the *index field* specify the set number, one of 128, within the cache. The high-order 21 bits are the *tag field*; these bits are compared with tags for each cache line in the indexed set, and they indicate whether a 16-byte cache line is stored for that physical address. The low-order 4 bits of the physical address select the byte within the cache line. Finally, a 4-bit valid field, one for each way within a given set, indicates whether the cached data at that physical address is currently valid.

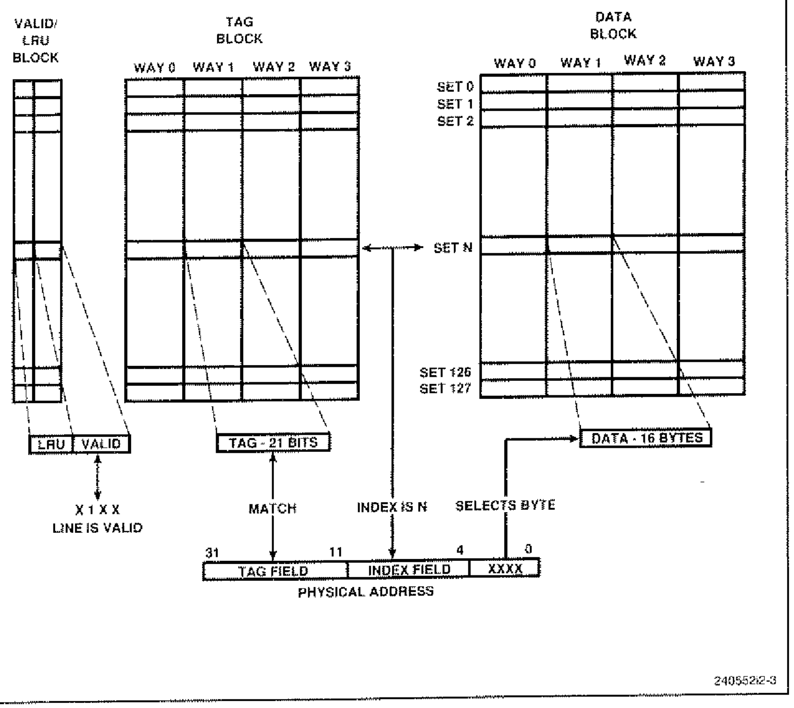


Figure 2-3. Cache Organization

2.3.2 Cache Updating

When a cache miss occurs on a read, the 16-byte block containing the requested information is written into the cache. Data in the neighborhood of the required data is also read into the cache, but the exact position of data within the cache line depends on its location in memory with respect to addresses divisible by 16.

Any area of memory is cacheable, but any page of memory can be declared not cacheable by setting a bit in its page table entry. When a read from memory is initiated on the bus, external logic can indicate whether the data may be placed in cache, as discussed in Chapter 3. If the read is cacheable, the processor attempts to read an entire 16-byte cache line.

The unit is a write-through cache. Cache line fills are performed only for read misses, never for write misses. When the processor is enabled for normal caching and write-through operation, every internal write to the cache (cache hit) not only updates the cache but is also passed along to the bus interface unit and propagated through the processor bus to memory. The only conditions under which data in the cache differs from the corresponding data in memory occur when a processor write cycle to memory is delayed by buffering in the bus interface unit, or when an external bus master alters the memory area mapped to the internal cache.

2.3.3 Cache Replacement

Replacement in the cache is handled by a pseudo-LRU (least recently used) mechanism. This mechanism maintains three bits for each set in the valid/LRU block, as shown in Figure 2-3. The LRU bits; are updated on each cache hit or cache line fill. Each cache line (four per set) also has an associated valid bit which indicates whether the line contains valid data. When the cache is flushed or the processor is reset, all of the valid bits are cleared. When a cache line is to be filled, a location for the fill is selected by simply finding any cache line which is invalid. If no cache line is invalid, the LRU bits select the line to be overwritten. Valid bits are not set for lines which are only partially valid.

Cache lines can be invalidated individually by a *cache line invalidation* operation on the processor bus. When such an operation is initiated, the cache unit compares the address to be invalidated with tags for the lines currently in cache and clears the valid bit if a match is found. A *cache flush* operation is also available. This invalidates the entire contents of internal cache unit.

2.3.4 Cache Configuration

Configuration of the cache unit is controlled by two bits in the processor's machine status register (CR0). One of these bits enables caching (cache line fills). The other bit enables memory write-through. The four configuration options are shown in Table 2-1. Chapter 3 gives details.

When caching is enabled, memory reads and instruction prefetches are cacheable. These transfers will be cached if external logic asserts the cache enable input in that bus cycle, and if the current page table entry allows caching. During cycles in which caching is disabled, cache lines will not be filled on cache misses. However, the cache remains active even though it is disabled for further filling. Data already in the cache will be used if it is still valid. Only when all data in the cache is flagged invalid, as happens in a cache flush, will all internal read requests be propagated as bus cycles to the external system.

When cache write-throughs are enabled, all writes, including those which are cache hits, are written through to memory. Invalidation operations will remove a line from cache if the invalidate address maps to a cache line. When cache write-throughs are disabled, an internal write request which is a cache hit will not cause a write-through to memory, and cache invalidation operations are disabled. With both caching and cache write-through

Table 2-1. Cache Configuration Options

Cache Enabled	Write-Through Enabled	Operating Mode
no	no	Cache line fills, cache write-throughs, and cache invalidations are disabled. This configuration allows the internal cache to be used as high-speed static RAM.
no	yes	Cache line fills are disabled, and cache write-throughs and cache invalidations are enabled. This configuration allows software to disable the cache for a short time, then re-enable it without flushing the original contents.
yes	no	INVALID
yes	yes	Cache line fills, cache write-throughs, and cache invalidations are enabled. The is the normal operating configuration.

disabled, the cache can be used as a high-speed static RAM. In this configuration, the only write cycles which are propagated to the processor bus are cache misses, and cache invalidation operations are ignored.

2.4 INSTRUCTION PREFETCH UNIT

When the bus interface unit is not performing bus cycles to execute an instruction, the instruction prefetch unit; uses the bus interface unit to prefetch instructions. By reading instructions before they are needed, the processor rarely needs to wait for an instruction prefetch cycle on the processor bus.

Instruction prefetch cycles; read 16-byte blocks of instructions;, starting at addresses numerically greater than the last-fetched instruction. The starting address is generated by the prefetch unit, which has a direct connection (not shown in Figure 2-1) to the paging unit. The 16-byte prefetched blocks are read into both the prefetch and cache units simultaneously. The prefetch queue; in the prefetch unit stores 32 bytes of instructions. As each instruction is fetched from the queue, the code part is sent to the instruction decode unit and (depending on the instruction) the displacement part is sent to the segmentation unit where it is used for address calculation. If loops are encountered in the program being executed, the prefetch unit gets copies of previously executed instructions from the cache.

The prefetch unit has the lowest priority for processor bus access. Assuming zero wait-state memory access, prefetch activity never delays execution. However, if there is no pending data transfer, prefetching may use bus cycles that would otherwise be idle. The prefetch unit is flushed whenever the next instruction needed is not in numerical sequence with the previous instruction—for example, during jumps, task switches, exceptions, and interrupts.

The prefetch unit will never access beyond the end of a code segment and it will never access a page that is not present. However, prefetching may cause problems for some hardware mechanisms. For example, prefetching may cause an interrupt when program execution nears the end of memory. To keep prefetching from reading past a given address, instructions should come no closer to that address than one byte plus one aligned 16-byte block.

2.5 INSTRUCTION DECODE UNIT

The instruction decode unit; receives instructions from the instruction prefetch unit and translates them in a two-stage process into low-level control signals and microcode entry points, as shown in Figure 2-1. Most instructions can be decoded at a rate of one per clock. Stage 1 of the decode, shown in Figure 2-2, initiates a memory access. This allows execution of a two-instruction sequence which loads and operates on data in just two clocks, as described above in Section 2.2.

The decode unit simultaneously processes instruction prefix bytes, opcodes, modR/M bytes, and displacements. The outputs include hardwired microinstructions to the segmentation, integer, and floating-point units. The unit is flushed whenever the instruction prefetch unit is flushed.

2.6 CONTROL UNIT

The control unit interprets the instruction word and microcode entry points received from the instruction decode unit. The control unit has outputs with which it controls the integer and floating-point processing units. It also controls segmentation because segment selection may be specified by instructions.

The control unit contains the processor's microcode. Many instructions have only one line of microcode, so they can execute in an average of one clock cycle. Figure 2-2 shows how execution fits into the internal pipelining mechanism.

2.7 INTEGER (DATAPATH) UNIT

The integer and datapath unit identifies where data is stored and performs all of the arithmetic and logical operations available in the 386 processor's instruction set, plus a few new instructions. It has eight 32-bit general-purpose registers, several specialized registers, an ALU, and a barrel shifter. Single load, store, addition, subtraction, logic, and shift instructions are executed in one clock.

Two 32-bit bidirectional buses connect the integer and floating-point units. These buses are used together for transferring 64-bit operands. The same buses also connect the processing units with the cache unit. The contents of the general purpose registers are sent to the segmentation unit on a separate 32-bit bus for generation of effective addresses.

2.8 FLOATING-POINT UNIT

The floating-point unit; executes the same instruction set as the 387 math coprocessor. The unit contains a push-down register stack and dedicated hardware for interpreting the 32-, 64-, and 80-bit formats specified in IEEE Standard 754. An output signal passed through to the processor bus indicates floating-point errors to the external system, which in turn can assert an input to the processor indicating that the processor should ignore these errors and continue normal operations.

2.9 SEGMENTATION UNIT

A segment is a protected, independent address space. Segmentation is used to enforce isolation among application programs, to invoke recovery procedures, and to isolate the effects of programming errors.

The segmentation unit translates a segmented address issued by a program, called a *logical address*, into an unsegmented address, called a *linear address*. The locations of segments in the linear address space are stored in data structures called *segment descriptors*. The segmentation unit performs its address calculations using segment descriptors and displacements (offsets) extracted from instructions. Linear addresses are sent to the paging and cache units. When a segment is accessed for the first time, its segment descriptor is copied into a processor register. A program can have as many as 16,383 segments. Up to six segment descriptors can be held in processor registers at any one time. Figure 2-4 shows the relationships between logical, linear, and physical addresses.

2.10 PAGING UNIT

The paging unit allows access to data structures larger than the available memory space by keeping them partly in memory and partly on disk. Paging divides the linear address space into 4-Kbyte blocks called *pages*. Paging uses data structures in memory called *page tables* for mapping a *linear address* to a *physical address*. Physical addresses are used by the cache and/or put on the processor bus. The paging unit also identifies problems, such as accesses to a page which is not resident in memory, and raises exceptions called *page faults*. On a page fault, the operating system has a chance to bring the required page into memory from disk. If necessary, it can free space in memory by sending some other page out to disk. If paging is not enabled, the physical address is identical to the linear address.

The paging unit includes a *translation lookaside buffer* (TLB) which stores the most recently used 32 page table entries. The TLB data structures are shown in Figure 2-5. The paging unit looks up linear addresses in the TLB. If the paging unit does not find a linear address in the TLB, the unit generates requests to fill the TLB with the correct physical address contained in a page table in memory. Only when the correct page table entry is in the TLB does the bus cycle take place. When the paging unit maps a page in

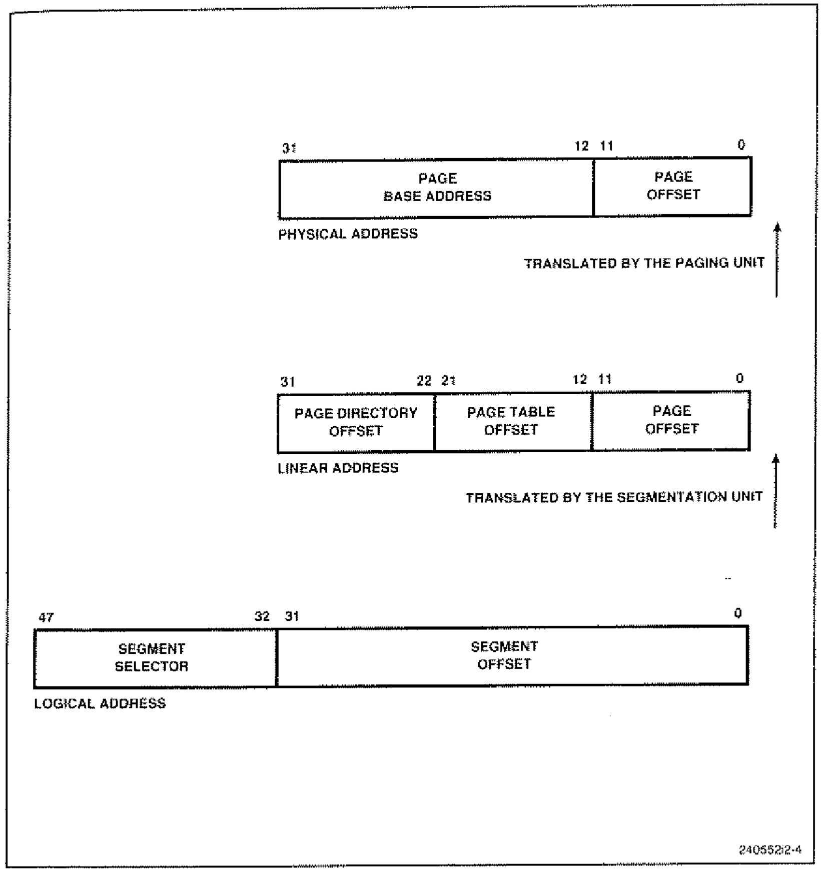


Figure 2-4. Segmentation and Paging Address Formats

the linear address space to a page in physical memory, it only maps the upper 20 bits of the linear address. The lowest 12 bits of the physical address come unchanged from the linear address.

Most programs access only a small number of pages during any short span of time. When this is true, the pages stay in memory and the address translation information stays in the TLB. In typical systems, 99% of the requests to access the page tables are satisfied by the TLB. The TLB uses a pseudo-LRU algorithm, similar to the cache, as a content-replacement strategy.

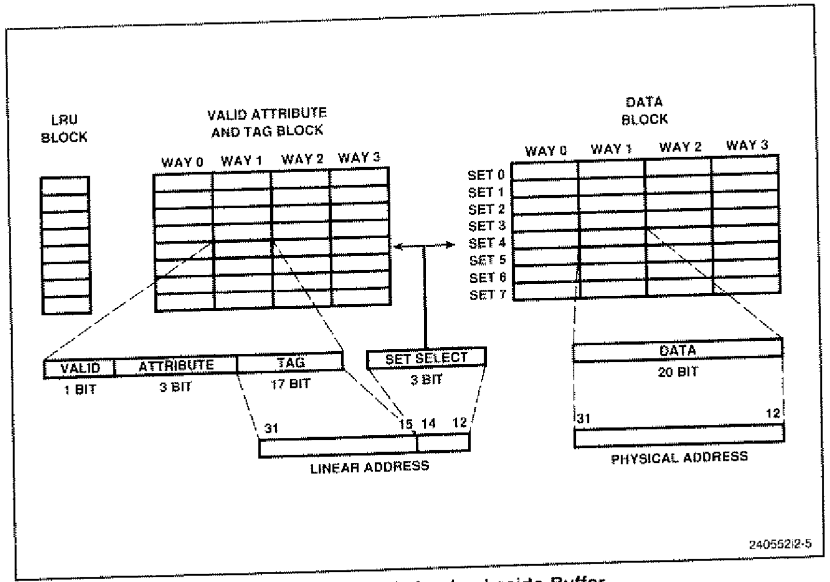


Figure 2-5. Translation Lookaside Buffer

The TLB is flushed whenever the page directory base register (CR3) is loaded. Page faults can occur during either a page directory read or a page table read. The cache can be used to supply data for the TLB, although this may not be desirable when external logic monitors TLB updates.

Unlike segmentation, paging is invisible to application programs and does not provide the same kind of protection against programs altering data outside a restricted part of memory. Paging is visible to the operating system, which uses it to satisfy application program memory requirements. For more information on paging and segmentation, see the *i486™ Programmer's Reference Manual*.

Processor Bus 3

CHAPTER 3
PROCESSOR BUS

3.1 OVERVIEW OF THE BUS

The processor bus is the set of pinout signals on the i486™ processor chip. It is the bus through which the processor communicates with other devices in the system. The signals on the bus are classed by their functions, which include bus control and arbitration, bus cycle definition and control, address and data, cache control, and floating-point error control.

The features of the processor bus include:

- Non-multiplexed 32-bit address and data buses.
- Single-frequency (1x) clock.
- Bus hold operations.
- Bus locking and pseudo-locking operations.
- Burst transfers (up to 16 bytes).
- Cacheable transfers.
- Support for internal and external cache consistency.
- Floating-point error handling.
- Maskable and non-maskable interrupts.
- Support for 16- and 8-bit peripherals.
- Support for 1-Mbyte 8086 address wrap-around.
- Parity generation and checking.

The way in which system designs use the processor bus has an important effect on performance. Typically, only a few devices are located on the bus – those which need fast communication with the processor, share compatible signals, and observe the basic constraint on use of bus bandwidth: at least 50% of processor-bus bandwidth should be reserved for the i486 processor. Devices placed on the bus might include a LAN coprocessor, an external (second-level) cache controller, or other similar device. In most systems, the processor bus interfaces with one or more system buses. This distributes bus traffic across greater bus bandwidth and provides greater flexibility for system expansion. The design of external buses need not conform to the signal set of the processor bus. Chapter 8 describes general approaches to system design, Chapter 7 describes system peripherals, and Chapter 9 describes interfaces to the MULTIBUS II backplane.

Write cycles dominate i486 processor bus activity. This is unlike most other systems in which read cycles dominate bus activity and can keep the processor waiting. With the i486 processor's internal cache, instruction prefetch unit, and support for burst transfers, any memory subsystem capable of sustaining a rate of one data transfer per clock cycle can form the basis of a high-performance system. Most of the processor's immediate needs for instructions and data will then be satisfied quickly from the internal cache and instruction prefetch queue, without having to perform cycles on the processor bus.

The processor bus can support multiple external caches. Cache consistency can be maintained between the processor's internal cache, external caches, and main memory. External cache can be requested to write its contents back to memory, or it can be flushed; individual cache lines in the internal cache can be selectively invalidated, or the entire internal cache can be flushed.

The 386 processor used address pipelining on the processor bus to minimize processor waiting time. In the i486 processor, burst reads into the on-chip cache are used, rather than address pipelining on the bus, to achieve high performance. This, together with the simpler 1x bus clock and more latitude in bus-cycle scheduling, results in simpler system logic.

Two processor inputs dynamically control bus size for interfacing 8- and 16-bit devices to the processor's data bus. There are no restrictions on byte or word alignment within doubleword boundaries, although data that is not aligned to doubleword boundaries requires more than the minimum number of bus cycles to transfer. The bus supports an emulation of the 8086 processor's 1-Mbyte address wrap-around.

The sections below first summarize, and later elaborate on, the use of processor bus signals, how the signals work together during bus cycles, and other matters relating to the processor bus.

3.1.1 Bus Cycles

Bus cycles implement the processor's interactions with the external system. The processor can drive two basic groups of bus cycles.

- *Data Transfer Cycles:*
  Prefetch (read) instructions from memory.
  Read data from memory.
  Read data from I/O.
  Write data to memory.
  Write data to I/O.
- *Other Cycles:*
  Interrupt acknowledgement.
  Halt (a special bus cycle).
  Shutdown (a special bus cycle).
  Cache flush (a special bus cycle).
  Cache write-back and flush (a special bus cycle).

Some of the cycles driven by the processor are, or can be, locked or pseudo-locked. External hardware can exercise a bus hold operation and drive its own cycles on the processor bus, including cache invalidation cycles into the processor.

From the viewpoint of external hardware, data can be transferred as doublewords, words, or bytes, depending on the bus size specified. From the processor's viewpoint, all transfers use the 32-bit data bus but some transfers have only certain bytes enabled. Bus cycles which transfer data are of two basic types:

- *Non-Burst Cycles*—These cycles transfer up to four bytes at a maximum rate of two clocks per data item (doubleword, word, or byte). When a single data item is transferred, it is a *single-transfer cycle*. When these single cycles are repeated in a series, they form a *multiple-cycle sequence*.
- *Burst Cycles*—The fastest way to transfer more than one item of data is with a burst cycle. These cycles transfer up to 16 bytes at a rate of one data item per clock cycle. They are designed for cacheable reads (each internal cache line holds 16 bytes) but they can also be used for long floating-point reads, segment table descriptor reads, and other types of transfers.

Transfers internal to the processor, such as reads from the internal cache, do not appear on the processor bus. However, writes to the cache always appear on the bus because the cache uses a write-through policy: all writes go to memory and they will only go to the internal cache if the addressed data is already stored in the cache.

The remaining types of bus cycles, aside from the data transfers discussed above, include interrupt acknowledgement and four special bus cycles. The details of interrupt acknowledgement cycles are given in Section 3.3.2. The four special bus cycles are described in Sections 3.3.3 (halt and shutdown), 3.4.4 (cache flush cycle), and 3.4.5 (cache write-back and flush cycle).

3.1.2 Overview of Signals and Control Cycles

Table 3-1 lists the signals on the processor bus. Tables 3-2 and 3-3 provide additional perspectives on the signals. Table 3-3 shows that certain input signals have internal pullup or pulldown resistors. These resistors will cause current to flow in these inputs, but the resistors should not be relied upon as the sole connection for an input. All unused inputs should be connected to an external pullup or pulldown.

Some signals on the processor bus have a dual use, one for normal operations and another for device testing; only their normal function is described here. The power supply pins are not included. The *i486*™ *Microprocessor Data Sheet* contains full details on the timing and electrical characteristics of all signals. This data sheet is the only authoritative source for timing and electrical information. The classification of signals in the data sheet differs somewhat from the classification shown in Figure 3-1—the signals, of course, are the same, but the viewpoint each reader may have of their functions can differ.

The text immediately following the tables summarizes the function of each signal. It also includes descriptions of the five bus cycles (halt, shutdown, cache flush, cache write-back, and interrupt acknowledge) that perform control functions very much like signals. The data transfer cycles, for which the bus fundamentally exists, are then described in Section 3.2.

6 Cache Subsystem

CHAPTER 6
CACHE SUBSYSTEM

6.1 INTRODUCTION

Caches are an important means of improving system performance. The i486™ DX microprocessor has an on-chip, unified code and data cache. The on-chip cache is used for both instruction and data accesses and operates on physical addresses. The i486 CPU has an 8-Kbyte cache which is organized in a 4-way set associative manner. To understand cache philosophy and the system advantages of a cache, many issues must be considered.

This chapter discusses the following related cache issues:

- Cache theory and the impact of caches on performance.
- The relationship between cache size and hit rates when using a single-level cache.
- Issues in mapping (or associativity) that arise when main memory is cached. Different cache configurations including direct-mapped, set associative, and fully associative. They are discussed along with the performance tradeoffs inherent to each configuration.
- The impact of cache line sizes and cache re-filling algorithms on performance.
- Write-back and write-through methods for updating main memory. How they maintain cache consistency and their impact on external bus utilization.
- Cache consistency issues that arise when a DMA occurs while the i486 CPU's cache is enabled. Methods that ensure cache and main memory consistency during cache accesses.
- Caches used in single versus multiple CPU systems.

6.2 CACHE MEMORY

Cache memories are high-speed memories that are placed between microprocessors and main memories. They keep copies of main memory that are currently in use to speed microprocessor access to requested data and instructions. When properly implemented, their access time can be three to eight times faster than that of main memory, and thus can reduce the overall access time. Caches also reduce the number of accesses to main memory DRAM which is important to systems with multiple CPU's which all access that same memory. This section introduces the cache concept and memory performance benefits provided by a cache.

6.2.1 What is a Cache?

A cache memory is a smaller high-speed memory that fits between a CPU and a slower main memory. Cache memories are important in increasing computer performance by reducing total memory latency. A cache memory consists of a directory (or tag), and a data memory. Whenever the CPU is required to read or write data it first accesses the tag memory and determines if a cache hit occurred, implying that the requested word is present in the cache. If the tags do not match, the data word is not present in the cache. This is called a cache miss. On a cache hit, the cache data memory allows a read operation to be completed more quickly from its faster memory than from a slower main memory access. The hit rate is the percentage of the accesses that are hits, and is affected by the size and organization of the cache, the cache algorithm used, and the program running. An effective cache system should maintain data in a way that increases the hit rate. Different cache organizations will be discussed later in this chapter. The main advantage of caches is that a larger main memory appears to have the high speed of a cache. For example, a zero-waitstate cache that has a hit rate of 90 percent will make main memory appear to be zero-waitstate memory for 9 out of 10 accesses.

Programs usually address memory in the neighborhood of recently accessed locations. This is called program locality or locality of reference and it is locality that makes cache systems possible. Code, data character strings, and vectors tend to be sequentially scanned items or items accessed repeatedly, and caches will help the performance in these cases. In some cases the program locality principle does not apply. Jumps in code sequences and context switching are some examples.

6.2.2 Why Add a Cache?

A cache can increase system performance at a reduced cost. Caches make main memory act as though it is performing at near-SRAM speed at a cost much less than a complete SRAM memory system. Caches are commonly used in high-speed 386™ CPU systems. A 386 DX CPU system with a 32-Kbyte cache and a main memory of 16 Mbytes is shown in Figure 6-1. A 32-Kbyte, direct-mapped cache using an 82385 cache controller has an 86 percent hit rate; it responds to the CPU in 0 waitstates (SRAM speeds) 86 percent of the time. This makes the 16 Mbytes of slower main memory appear to be 16 Mbytes of SRAM in 86 percent of its reads.

The 386 DX performance benefits from an 82385 cache about 25-35% compared to DRAM alone. The i486 CPU, however, has an 8K internal cache. An external cache for the i486 CPU, called a second-level cache, will offer anywhere from no to moderate performance increases. The on-chip, 8K cache is sufficient for most applications, but thrashes during larger, memory-intensive or multi-process applications. A large, second-level cache can capture the data that misses the internal cache and provide near-SRAM speed response. The effectiveness of the second-level cache widely varies and depends on the application being executed and the main memory speed.

6.3 CACHE TRADEOFFS

Cache efficiency is the cache's ability to keep the most frequently used code and data used by the microprocessor, and it is measured in terms of the hit rate. Another indication of cache efficiency is system performance; this is the speed in which the microprocessor can perform a certain task and is measured in effective bus cycles. An efficient cache reduces external bus cycles and enhances overall system performance. Hit rates are discussed in the next section.

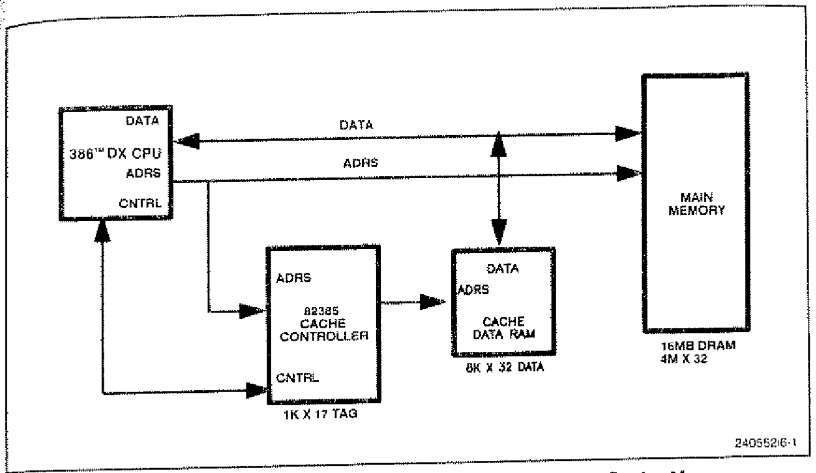


Figure 6-1. A Typical 386™ DX CPU System with an 82385 Cache Memory

Factors that can affect a cache's performance are:

- Size: Increasing the cache size allows more items to be contained in the cache. Cost is increased, however, and a larger cache cannot operate as quickly as a smaller one.
- Associativity (discussed in Section 6.2.2): Increased associativity increases the cache's hit rate but also increases its complexity and reduces its speed.
- Line Size: The amount of data the cache must fetch during each cache line replacement (every miss) affects performance. More data takes more time to fill a cache line, but then more data is available and the hit rate increases.
- Write-Back and Write Posting: The ability to write quickly to the cache and have the cache then write to the slower memory increases performance. Implementing these types of caches can be very complex, however.
- Features: Adding features such as write-protection (to be able to cache ROM memory), bus watching, and multiprocessing protocols can speed a cache but increases cost and complexity.
- Speed: Not all caches return data to the CPU as quickly as possible. It is less expensive and complex to use slower cache memories and cache logic. Intel's 82385 and 485Turbocache Module are, however, the fastest possible solutions for the 386 DX and i486 microprocessor.

6.3.1 Cache Size and Performance

Hit rates for various first-level cache configurations are shown in Table 6-1. These statistics are conservative because they illustrate the lowest hit rates generated by analyzing several mainframe traces. The hit rates are not absolute quantities, and the hit

Table 6-1. First-Level Cache Hit Rates

Cache Configuration			Hit Rate
Size	Associativity	Line Size	
1K	direct	4 bytes	41%
8K	direct	4 bytes	73%
16K	direct	4 bytes	81%
32K	direct	4 bytes	86%
32K	2-way	4 bytes	87%
32K	direct	8 bytes	91%
64K	direct	4 bytes	88%
64K	2-way	4 bytes	89%
64K	4-way	4 bytes	89%
64K	direct	8 bytes	92%
64K	2-way	8 bytes	93%
128K	direct	4 bytes	89%
128K	2-way	4 bytes	89%
128K	direct	8 bytes	93%

rate of a particular configuration is software dependent. However, the table allows a meaningful comparison of the various cache configurations. It also indicates the degree of hardware complexity needed to arrive at a particular cache efficiency. Table 6-1 presents direct-mapped, 2-way, and 4-way set associative caches which are all discussed in the next section.

Program behavior is another important factor in determining cache efficiency. If a program uses a piece of data only once, then the cache may spend all its time thrashing or replacing itself with new data from memory. This is common in vector processing. The processor receives no added efficiency from the cache as main memory is being requested frequently. In such instances, the user can consider mapping the data entries as noncacheable.

Cache system performance can be calculated based on the main memory access time, the cache access time, the miss rate, and the write cycle time.

Cs is defined as the ratio of the cache system access time to the main memory access time. Cs is a dimensionless number but provides a useful measure of the cache performance.

$$Ca = (1-M)Tc + MTm$$
$$Cs = Ca/Tm = (1-M)(Tc/Tm) + M = (1-M)Cm + M$$

where:

Ca	= average cache system cycle time averaged over reads and writes
Tc	= cache cycle time
Tm	= main memory cycle time
M	= miss rate = 1-hit rate
Cs	= cache system access time as a fraction of main memory access time
Cm	= cache memory access time as compared to main memory cycle time

If the cache always misses then M = 1 and Cm = 1, and the main memory access is equal to the effective access time of the cache. If the cache is infinitely fast, then Cm is equal to the miss rate. Because the cache access time is finite, the cache system access time approaches the cache access time as the miss rate approaches zero.

While the above discussion applies to read operations it can be easily extended to write operations, which also affect system performance. When memory has to be written to, the CPU has to wait for the completion of the write cycle before proceeding to the next instruction. In a buffered memory system, where posted writes occur, data can be loaded in a register, and the memory can be updated later. This allows the CPU to begin the next cycle without being delayed by the main memory write access time. Both these memory updating techniques are discussed later in this chapter.

6.3.2 Associativity and Performance Issues

Data and instructions are written into the cache by a function that maps the main memory address into a cache location. The placement policy determines the mapping function from the main memory address to the cache location. There are four policies to consider: fully associative, direct-mapped, set associative, and sector buffering.

Fully Associative: A fully associative cache system provides maximum flexibility in determining which blocks are stored in the cache at any time. Ideally the blocks of words in the cache would contain the main memory locations needed most by the processor regardless of the distance between the words in main memory. The size of a block in the cache is also known as the line size, and corresponds to the width of a cache word. For example, a block can be eight bytes for a 32-bit processor, in which case two doublewords are accessed each time the cache line is filled. In the example shown in Figure 6-2, the block size is one doubleword.

Because there is no single relationship between all of the addresses in the 64 blocks, the cache would have to store the entire address of each block. When the processor requests data, the cache controller would have to compare the address with each of the 64 addresses in the cache for a match condition. This organization, shown in Figure 6-2 is called fully associative.

Direct Mapped: In a direct mapped cache, the simplest of the three policies, only one address comparison is required to determine if the requested word is in the cache. This is because each block in the cache maps to only one location in the cache. A direct mapped cache address has two parts: a cache index field, which specifies the block's location in the cache, and a tag field that distinguishes blocks within a particular cache location.

For example, consider a 64-Kbyte direct mapped cache that contains 16K 32-bit locations and caches 16 Mbytes of memory. The cache index field must include 14 bits to select one of the 16-Kbyte blocks in cache plus two bits to decode one of the four byte enables. The tag field must be eight bits wide to identify one of the 256 blocks that can

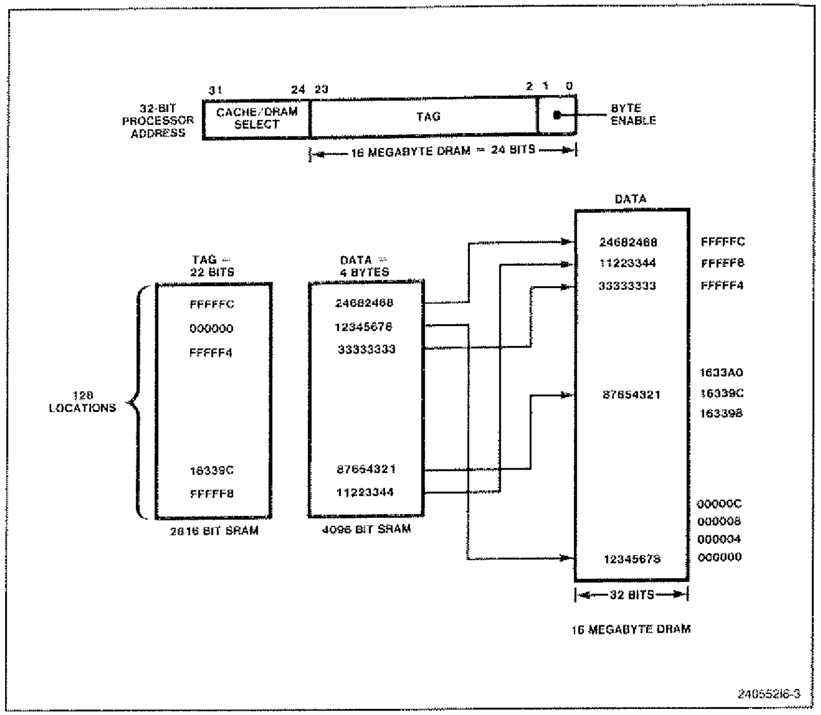


Figure 6-2. A Fully Associative Cache Organization

occupy the selected cache location. The most significant eight bits of the address are decoded to select the cache subsystem from other memories in the memory space. The direct-mapped cache organization is shown in Figure 6-3.

If the processor requests data at FFFFF8, then the first step is to send the most significant 14 bits of FFF8 to the cache tag RAM. If the tag field stored at FFF8 is FF (as shown in the diagram), then a hit has occurred and the data word "B" is sent to the CPU. If the requested word has 020004, then the tags would not match. In this case the tag RAM would be updated with the value 02 corresponding to the index 0004, and the data "D" would be replaced by the word at location 020004.

If the processor accesses locations that have the same index bits, then the cache would have to be updated constantly. This type of program behavior is infrequent, however, so a direct mapped cache may provide acceptable performance at a lower cost when compared to a fully associative cache memory.

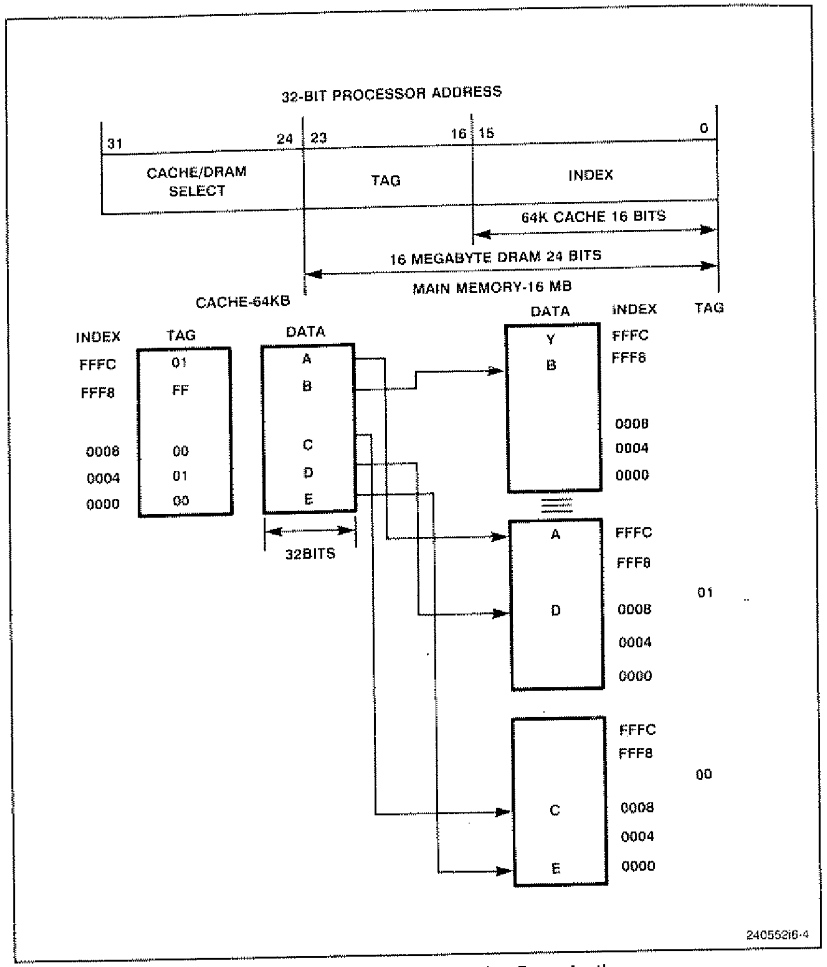


Figure 6-3. Direct Mapped Cache Organization

Set Associative: The set-associative cache is a compromise between the fully associative and direct-mapped caches. The set-associative cache has more than one set and it is equivalent to several direct mapped caches operating in parallel. For each cache index there are several block locations allowed, and the block can be placed in any set or retrieved from any set. Figure 6-4 shows a two-way set associative cache memory.

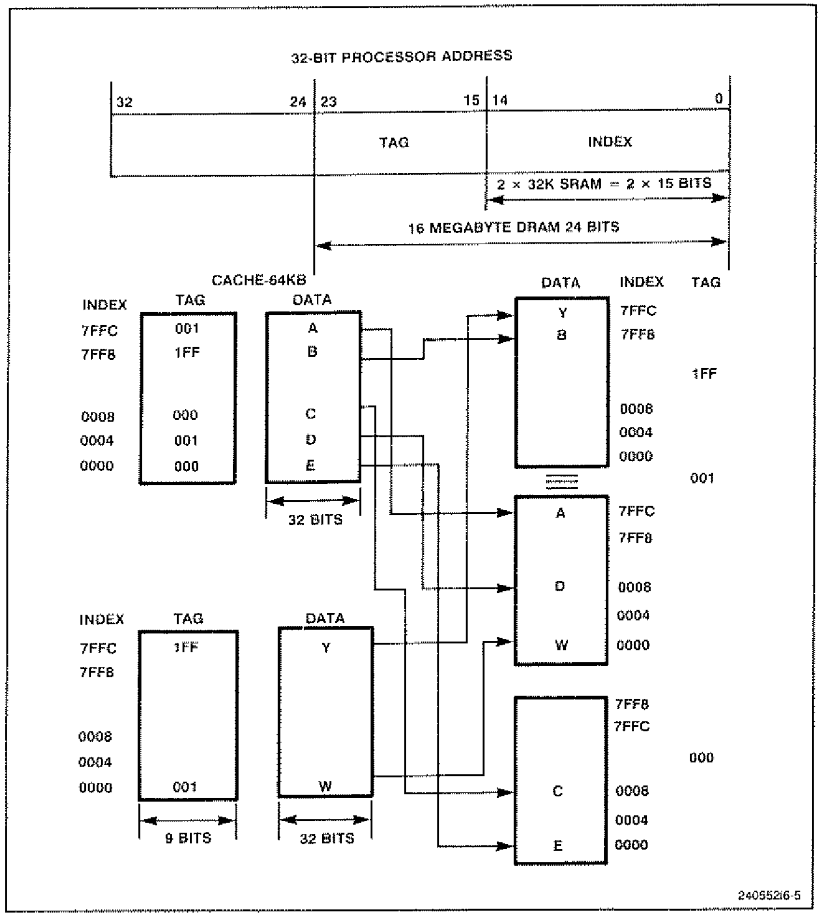


Figure 6-4. Two-Way Set Associative Cache Organization

Given an equal amount of cache memory as in the direct mapped example, the set associative cache has half as many locations, and the extra address bit becomes part of the tag field. Because the set-associative cache has several places for a block with the same cache index, the hit rate is increased. The set associative cache performs more efficiently than a direct mapped cache, but it needs a wider tag field and additional logic to determine which set should receive the data. This function is determined by the replacement policy, which is covered later in this section.

Sector Buffering: Another cache configuration uses a sector buffer and is sometimes called a sub-block cache. The cache is a number of sectors, and the sectors in turn are a number of blocks. Each block can have its own valid bit, but only one tag address exists per sector. When a word is accessed whose sector is in the cache but the block is not, then the block is fetched from the main memory. Sector buffering has its own tradeoffs associated with miss ratios and bus utilization. Having smaller blocks increases the miss ratio, but reduces the number of external bus accesses. Conversely, having a large number of blocks increases the hit ratio but also increases the external bus utilization. Figure 6-5 shows the cache organization in sector buffering.

The i486 CPU's on-board cache is organized 4-way set associative with a line size of 16 bytes. The 8-Kbyte cache is organized as four 2-Kbyte sets. Each 2-Kbyte set is comprised of one hundred and twenty-eight 16 byte-lines. Figure 6-6 shows the cache organization. Because the cache is on-chip, the user can achieve an extremely high hit rate with the 4-way associativity. The cache is transparent so that the i486 CPU remains software compatible with its non-cache predecessors.

6.3.3 Block/Line Size

As mentioned earlier, block size is an important consideration in cache memory design. Block size is also referred to as the line size or the width of the cache data word. The block size may be larger than the word, and this can impact the performance as the cache may be fetching and storing more information than the CPU needs.

As the block size increases, the number of blocks that fit in the cache are reduced. Because each block fetch overwrites the older cache contents, some blocks are overwritten shortly after being fetched. In addition, as block size increases, additional words are fetched with the requested word. Because of program locality the additional words are less likely to be needed by the processor.

TAG 1	BLOCK 1.1	BLOCK 1.2	BLOCK 1.3	. . .	BLOCK 1.N
TAG 2	BLOCK 2.1	BLOCK 2.2	BLOCK 2.3	. . .	BLOCK 2.N
. . .	. . .	. . .	. . .	. . .	. . .
TAG M	BLOCK M.1	BLOCK M.2	BLOCK M.3	. . .	. . .

TAG PER SECTOR BLOCKS PER SECTOR

Figure 6-5. Sector Buffer Cache Organization

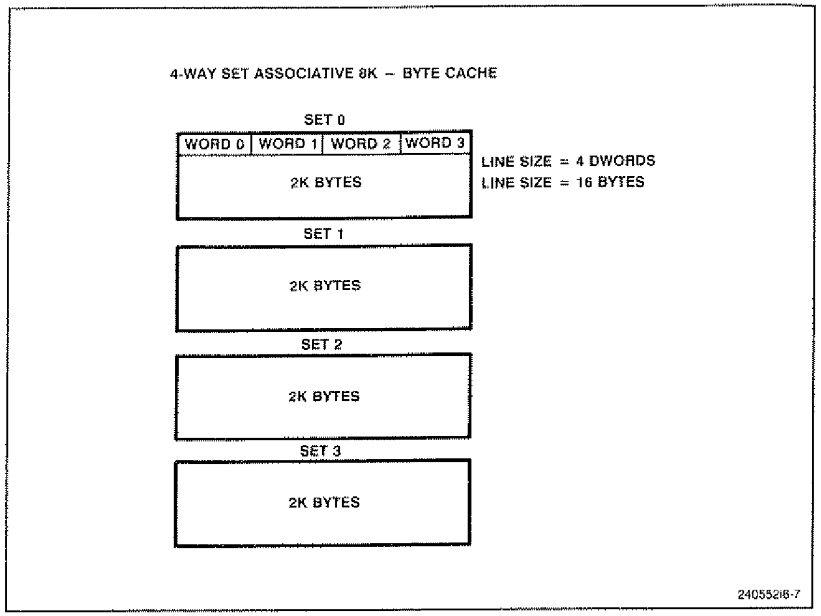


Figure 6-6. The Cache Data Organization for the On-Chip i486™ CPU's Cache

If a cache is refilled with 4 or 8 CPU words on a miss then the performance improves dramatically over a cache size that employs single word refills. Those extra words that were read into the cache, because they are subsequent words and by the sequential nature of programs, will most likely be hits in subsequent cache accesses. As well, the cache refill algorithm is a significant performance factor in systems where the delay in transferring the first word from the main memory is long but in which several subsequent words can be transferred in a shorter time. This situation applies when using page mode accesses in dynamic RAMs, and the initial word is read after the normal access time, while subsequent words can be quickly accessed by changing only the column addresses. Taking advantage of this situation while selecting the optimum line size can greatly increase cache performance.

6.3.4 Replacement Policy

In a set-associative cache configuration, a replacement policy is needed to determine which set should receive new data when the cache is updated. There are three common approaches for choosing which block (or single word) within a set will be overwritten. These are the least recently used (LRU) method, the first-in first-out (FIFO) method, and the random method.

In the LRU method, the set that was least recently accessed is overwritten. The control logic must maintain least recently used bits and must examine the bits before an update occurs. In the FIFO method, the cache overwrites the block that is resident for the longest time. In the random method, the cache arbitrarily replaces a block. The performance of the algorithms depends on the program behavior. The LRU method is preferred because it provides the best hit rate.

6.4 UPDATING MAIN MEMORY

When the processor executes instructions that modify the contents of the cache, changes have to be made in the main memory as well, otherwise, the cache is only a temporary buffer and it is possible for data inconsistencies to arise between the main memory and the cache. If only one of the two, the cache or the main memory, is altered and the other is not, two different sets of data become associated with the same address. A potential situation of incorrect or stale data is shown in Figure 6-7. There are two general approaches to updating the main memory. The first is the write-through method, and the second is the write-back, also known as copy-back method. Memory traffic issues are discussed for both the methods.

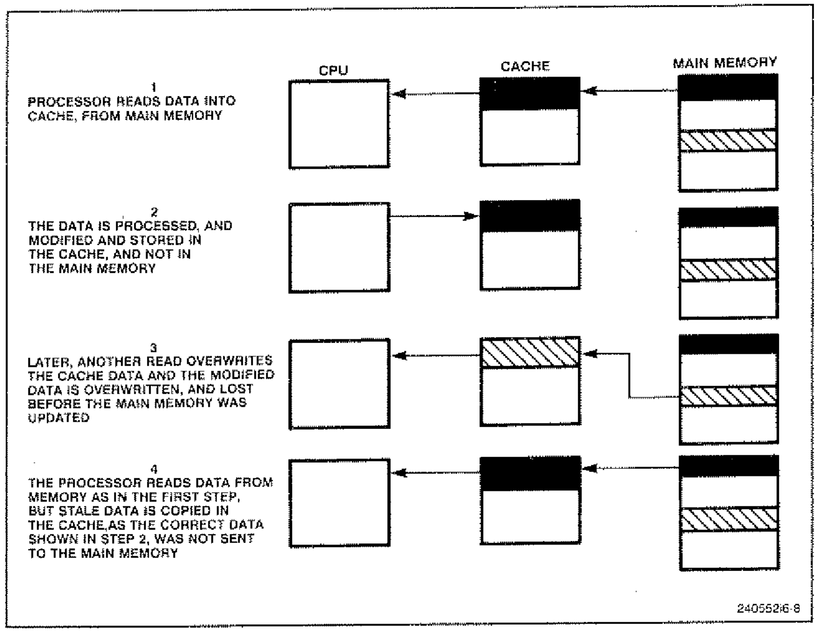

CPU
CACHE
MAIN MEMORY
1
PROCESSOR READS DATA INTO CACHE, FROM MAIN MEMORY
2
THE DATA IS PROCESSED, AND MODIFIED AND STORED IN THE CACHE, AND NOT IN THE MAIN MEMORY
3
LATER, ANOTHER READ OVERWRITES THE CACHE DATA AND THE MODIFIED DATA IS OVERWRITTEN, AND LOST BEFORE THE MAIN MEMORY WAS UPDATED
4
THE PROCESSOR READS DATA FROM MEMORY AS IN THE FIRST STEP, BUT STALE DATA IS COPIED IN THE CACHE,AS THE CORRECT DATA SHOWN IN STEP 2, WAS NOT SENT TO THE MAIN MEMORY
240552i6-8

Figure 6-7. Stale Data Problem in the Cache/Main Memory

6.4.1 Write-Through and Buffered Write-Through Systems

In a write-through system, data is written to the main memory immediately after or while it is written into the cache. As a result, the main memory always contains valid data. The advantage to this approach is that any block in the cache can be overwritten without data loss, while the hardware implementation remains fairly straightforward. There is a memory traffic tradeoff, however, because every write cycle increases the bus traffic on a slower memory bus. This can create contention by multiple bus masters for use of the memory bus. Even in a buffered write-through scheme, each write will eventually go to memory. Thus, bus utilization for write cycles is not reduced by using a write-through or buffered write-through cache.

Users sometimes adopt a buffered write-through approach in which the write accesses to the main memory can be buffered with a N-deep pipeline. A number of words are stored in pipelined registers, and will subsequently be written to the main memory. The processor can begin a new operation before the write operation to main memory is completed. If a read access follows a write access, and a cache hit occurs, then data can be accessed from the cache memory while the main memory is updated. If the N-deep pipeline is full the processor must wait if another write access occurs and the main memory has not been as yet been updated. A write access followed by a read miss also requires the processor to wait as the main memory has to be updated before the next read access.

Pipeline configurations must account for multiprocessor complications when another processor accesses a shared main memory location which has not been updated by the pipeline. This means the main memory hasn't been updated, and the memory controller must take the appropriate action to prevent data inconsistencies.

6.4.2 Write-Back System

In a write-back system, the processor writes data into the cache and sets a "dirty bit" which indicates that a word had been written into the cache but not into the main memory. The cache data is written into the main memory at a later time and the dirty bit is cleared. Before overwriting any word or block in the cache, the cache controller looks for a dirty bit and updates the main memory before loading the cache with the new data into the cache.

A write-back cache accesses memory less often than a write-through cache because the number of times that the main memory must be updated with altered cache locations is usually lower than the number of write accesses. This results in reduced traffic on the main memory bus.

A write-back cache can offer higher performance than a write-through cache if writes to main memory are slow. The primary use of the a write-back cache is in a multiprocessing environment. Since many processors must share the main memory, a write-back cache may be required to limit each processor's bus activity, and thus reduce contention for main memory. It has been shown that in a single-CPU environment with up to 4 clock memory writes, there is no significant performance difference between a write-through and write-back cache.

There are some disadvantages to a write-back system. The cache control logic is more complex because addresses have to be reconstructed from the tag RAM and the main memory has to be updated along with the pending request. For DMA and multiprocessor operations, all locations with an asserted dirty bit must be written to the main memory before another device can access the corresponding main memory locations.

6.4.3 Cache Consistency

Write-through and write-back systems require mechanisms to eliminate the problem of stale main memory in a multiprocessing system or in a system with a DMA controller. If the main memory is updated by one processor then the cache data maintained by another processor can contain stale data. A system that prevents the stale data problem is said to maintain cache consistency. There are four methods commonly used to maintain cache consistency: snooping (or bus watching), broadcasting (or hardware transparency), non-cacheable memory designation, and cache flushing.

In snooping, cache controllers monitor the bus lines and invalidate any shared locations that are written by another processor. The common cache location is invalidated and cache consistency is maintained. This method is shown in Figure 6-8.

In broadcasting/hardware transparency, the addresses of all stores are transmitted to all the other caches so that all copies are updated. This is accomplished by routing the accesses of all devices to main memory through the same cache. Another method is by copying all cache writes to main memory and to all of the caches that share main memory. A hardware transparent system is shown in Figure 6-9.

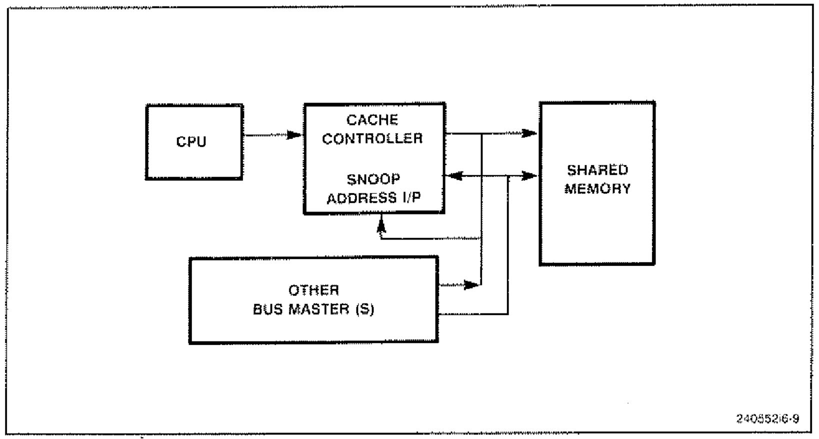


Figure 6-8. Bus Watching/Snooping for Shared Memory Systems

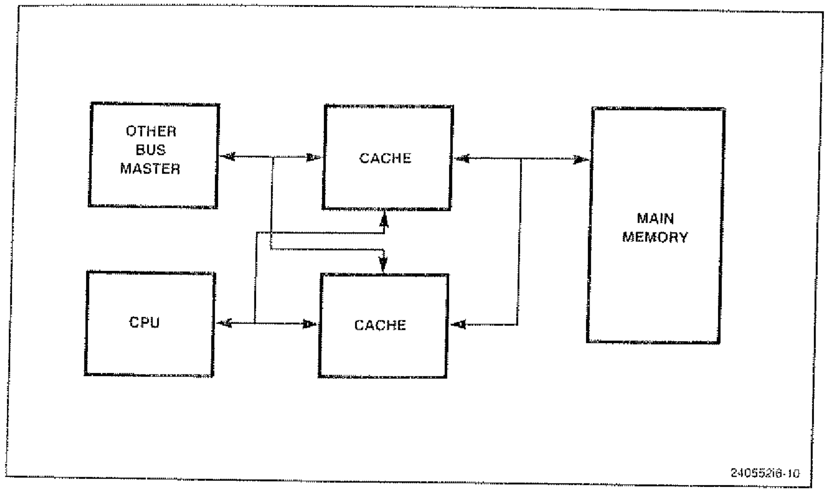


Figure 6-9. Hardware Transparency

In non-cacheable memory systems, all shared memory locations are considered non-cacheable. In such systems, access to the shared memory is never copied in the cache, and the cache never receives stale data. This can be implemented with chip select logic or with the high order address bits. Figure 6-10 shows non-cacheable memory.

In cache flushing, all cache locations with set dirty bits are written to main memory (for write-back systems), then cache contents are cleared. If all of the devices are flushed before another bus master writes to shared memory, cache consistency is maintained.

Combinations of various cache coherency techniques may be used in a system to provide an optimal solution. A system may use hardware transparency for time critical I/O operations such as paging, and it may partition the memory as non-cacheable for slower I/O operations such as printing.

6.5 CACHE AND DMA OPERATIONS

Some of the issues related to cache consistency in systems employing DMA have already been covered in the preceding section. Because a DMA controller or other bus master can update main memory there is a possibility of stale data in the cache. The problem can be avoided through snooping, cache transparency, and non-cacheable designs.

In snooping, the cache controller monitors the system address bus, and invalidates cache locations that will be written to during a DMA cycle. This method is advantageous in that the processor can access its cache during DMA operations to main memory. Only a "snoop hit" causes an invalidation cycle (or update cycle) to occur.

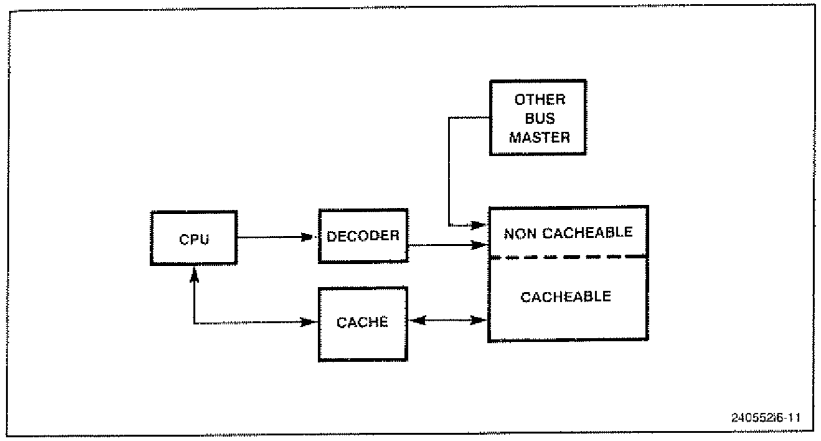


Figure 6-10. Non-Cacheable Shared Memory

In cache transparency, memory accesses through the CPU and the DMA controller are directed through the cache, requiring minimal hardware. However, the main disadvantage is that while a DMA operation is in progress the CPU bus is placed in HOLD. The concurrency of CPU/cache and DMA controller/main memory operations is cannot supported.

In non-cacheable designs, a separate dual-ported memory can be used as the non-cacheable portion of the memory, and the DMA device is tightly coupled to this memory. In this way the problem of stale data cannot occur.

In all of the approaches, the cache should be made software transparent so that DMA cycles do not require special software programming to ensure cache coherency.

6.6 CACHES FOR SINGLE VERSUS MULTIPLE CPU SYSTEMS

In single CPU systems, a write-through cache is an ideal cache solution. Write-through caches solve consistency issues, may be designed to be a plug-in option, and are less-expensive. For example, the 485Turbocache Module is a write-through, optional, expandable, second-level cache designed for the i486 CPU. The main drawback of a write-through cache is its inability to reduce main memory utilization for write cycles. However, this is not as critical a consideration to single CPU designs.

Memory bus utilization in multiple CPU systems is, perhaps, the most important performance consideration. In this type of system, a cache should have a very high hit rate for both reads and writes. Accesses to main, shared memory must be minimized. Write-back caches are best-suited for these multiprocessor environments. A write-back cache will, however, be more complex in its architecture and coherency mechanisms.

6.7 INTRODUCTION TO THE 485TURBOCACHE MODULE EXTERNAL CACHE AND THE MEMORY HIERARCHY

The 485Turbocache Module is a high-performance, optional, write-through, second-level cache that was designed specifically for the i486 CPU. It can be used in standalone mode or a cascaded mode to expand the cache depth. It increases the overall system performance by reducing the average number of wait states for memory reads. The 485Turbocache Module provides the highest performance for both the first word requested by the i486 CPU as well as the subsequent burst cycles. For the best performance at 25 and 33 MHz, a second-level cache is required. The 485Turbocache Module provides the best price/performance ratio for 25 and 33 MHz i486 CPU system designs.

The 485Turbocache Module is a 64-Kbyte or 128-Kbyte, 2-way, set associative cache with a line size of four doublewords. Each set is organized as 8K × 32 or 16K × 32. The 485Turbocache Module also contains on-board tag memory and comparator logic to provide a complete cache solution.

With a good memory design, the on-chip cache of the i486 CPU limits the performance increase that a second-level cache will offer. Additionally, some applications, such as Lotus 1-2-3, benefit less than 5 percent from an external cache, while others, such as Excel, benefit greater than 30 percent. Performance is extremely application sensitive as an application may or may not operate effectively within the confines of the on-chip 8K cache.

6.7.1 An i486 Processor System

A typical i486 microprocessor system is shown in Figure 6-11. The i486 processor has a local bus that consists of address, data and control buses. These buses are either buffered, registered or latched to comprise the system bus.

The memory subsystem is made up of DRAMs, SRAMs, and EPROMs. Main memory accesses are usually addressed to a DRAM subsystem, however the I/O subsystem can communicate with the i486 CPU, and it can also communicate with the memory subsystem during DMA operations.

Cache consistency must be maintained whenever main memory accesses occur during DMA operations. Bus snooping and validation logic can monitor the bus to detect memory writes that may be initiated by other bus masters. If such writes are detected, portions of the processor and the 485Turbocache Module second-level cache may have to be invalidated. Both the i486 CPU and the 485Turbocache Module have mechanisms that can invalidate cache entries.

The 485Turbocache Module is closely coupled to the i486 processor. The address, data, and control signals are connected to the processor's local bus, and 485Turbocache Module control signals interface to the system bus as well. The system bus control signals interface to the processor and the 485Turbocache Module in a similar manner. This allows the second-level 485Turbocache Module cache to be implemented into an i486 processor system with ease.

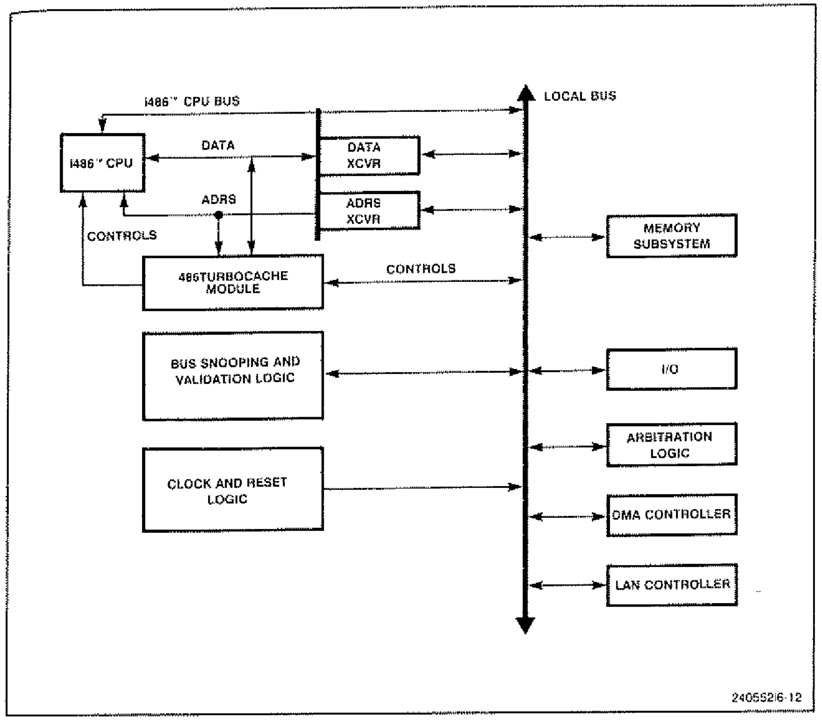
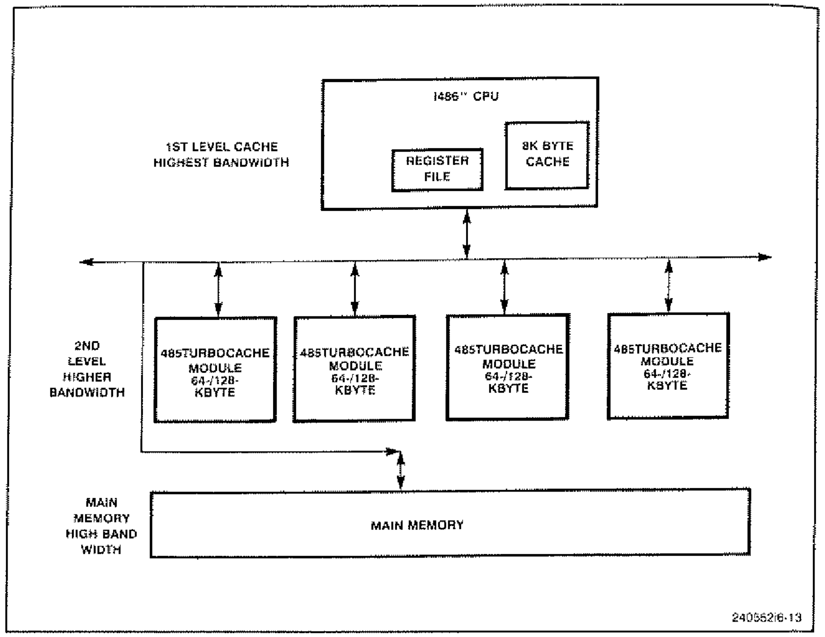


Figure 6-11. A Typical i486™ CPU System

6.7.2 The Memory Hierarchy and Advantages of a Secondary Cache

The i486 CPU has a high-speed register set and on-chip cache and these are accorded the first level of memory hierarchy. Instructions can be executed in a single clock, and at an average cycles per instruction rate of 1.8 (CPI). The next level of hierarchy is accorded to the secondary cache, which can consist of one or more 485Turbocache Modules. These sustain a high level of performance by supporting the fastest possible memory accesses, requiring only two clock cycles for the first read and one clock cycle for each of the subsequent three reads in a burst cycle. System performance degrades if main memory accesses are required. However, with the on-chip first-level cache and the external second-level cache, the number of main memory read accesses is reduced considerably. Figure 6-12 shows the memory hierarchy in a typical i486 processor system.

Because the i486 microprocessor internal cache is so efficient, most external CPU bus cycles are DRAM page misses. A second-level cache improves the bus latency problem, as data is available a large percentage of the time from the cache for read operations. A



i486™ CPU
1ST LEVEL CACHE HIGHEST BANDWIDTH
REGISTER FILE
8K BYTE CACHE
2ND LEVEL HIGHER BANDWIDTH
485TURBOCACHE MODULE 64-/128-KBYTE
485TURBOCACHE MODULE 64-/128-KBYTE
485TURBOCACHE MODULE 64-/128-KBYTE
485TURBOCACHE MODULE 64-/128-KBYTE
MAIN MEMORY HIGH BAND WIDTH
MAIN MEMORY
240552i6-13

Figure 6-12. i486™ Processor System Memory Hierarchy

large main memory can have an access time of six to eight cycles on a page miss. On page hits data can be provided in three or four cycles.

The 485Turbocache Module has a write protection feature that allows the cacheing of BIOS which greatly improves BIOS function performance. The i486 CPU can write-protect code on a page basis, but not in real mode. Common BIOS functions which can be accelerated by cacheing are extended memory accesses, video accesses, disk copies, and LAN card code. The 485Turbocache Module also can support 1-clock bursts, and supply the i486 CPU data at the fastest possible rate. Its 2-way set-associative configuration allows a main memory location to have the possibility of being present in one of two sets improving its hit rate. See Section 6.5.5 for 485Turbocache Module performance measurements.

6.7.3 485Turbocache Module Architecture

Figure 6-13 is an architectural block diagram of the 485Turbocache Module. The 485Turbocache Module has control signals that interface with the i486 processor, a main memory controller, and a bus controller.

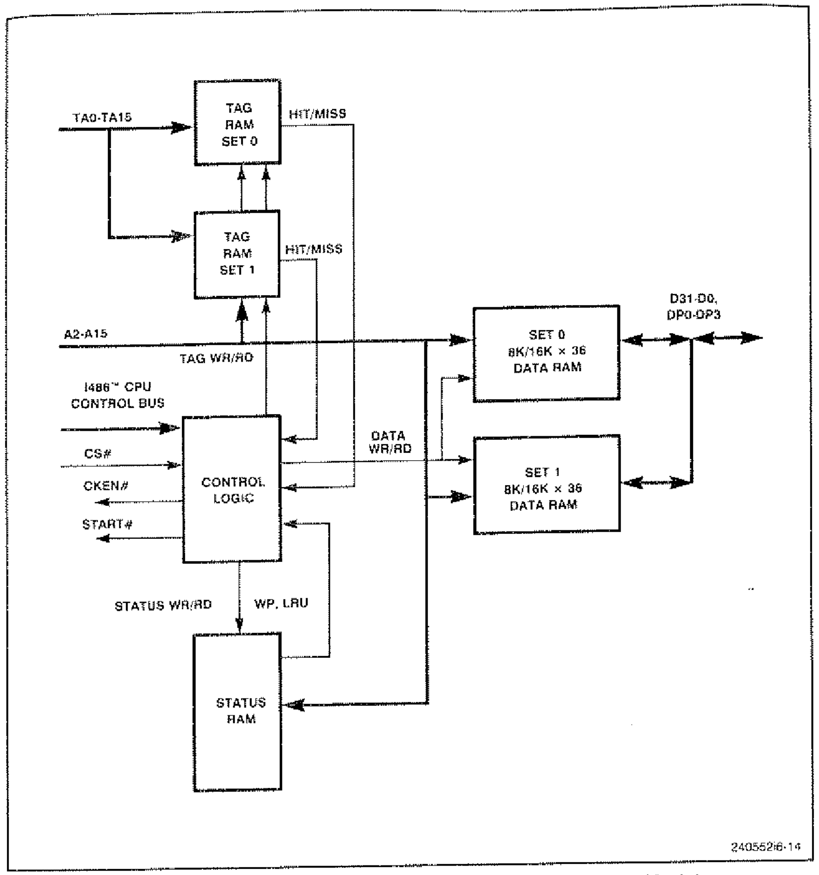


Figure 6-13. Internal Block Diagram of the 485Turbocache Module

The 485Turbocache Module is organized as two sets of 8K or 16K doublewords. When the cache is updated, four doublewords are accessed, making one of the sets is equivalent to 2K or 4K locations containing four doublewords. During a cache read operation, bits A2-A15 address both tag RAM sets. Tag addresses, TA0-TA15 are compared to the previously stored tags. If there is a hit condition, when the tags match, then data from the appropriate set is placed on the data bus. If there is a miss, then the START# signal is asserted to initiate a main memory access. On a cache read miss, four doublewords are read to the cache. Write operations update the tag RAMs and data RAMs only if a cache hit occurs.

The data RAM section has two banks of memory, and the data from one bank is selected and placed on the output data bus. The Data RAM is organized as two sets of 8K × 36 or 16K × 36. Parity bits are treated as additional data bits, and parity generation and checking is done externally.

Cache memory is expanded by simply adding more caches to any required depth. Chip select is decoded to select which cache slice should be activated.

The status RAM stores information on the least recently used (LRU) bit and the write protect (WP) bit. The LRU bit is used to track the least recently used set for overwriting during cache updates. The write protect bit is used to inhibit writes to particular locations, and can be set during cache line fills.

The control logic unit has the interface signals for the i486 CPU, a main memory controller, and a bus controller. 485Turbocache Modules can be cascaded using the CS# signal, to produce one larger cache. Internally, the control logic unit interfaces to the status RAM and the tag RAM to determine which bank will be accessed. It also provides the controls used to update the data RAM for burst cycles, in the same manner that is used to update the i486 CPU's cache.

6.7.4 System Overview

The i486 CPU, cache subsystem, main memory controller, and bus controller are shown in Figure 6-14. Multiple 128-Kbyte 485Turbocache Modules are cascaded to produce a 256-Kbyte cache or a 512-Kbyte cache. The 485Turbocache Module provides a START# signal to the Memory Controller for a read miss and for all write operations. The

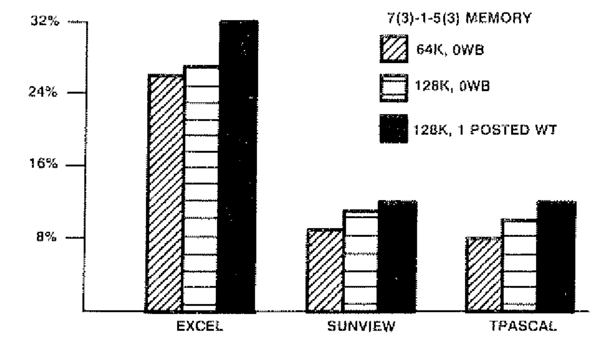


Figure 6-14. 485Turbocache Module Performance

START# signal is not asserted for I/O cycles. The memory controller provides an SKEN# signal, just like the i486 CPU's KEN# signal, to indicate whether an accessed word is cacheable.

The 485Turbocache Module has all of the required control signals needed to interface with the i486 CPU, and can be connected to a main memory controller or a bus controller with ease.

6.7.5 Performance

Figure 6-14 shows the performance boost that the 485Turbocache Module gives to various applications. Also included is the gain a single write buffer (posted write) will offer. The graphs were obtained by simulation of the i486 CPU with actual traces of each application. The simulator modeled a 128K 485Turbocache Module acting with a 7(3)-1-5(3) (reads take 7 clocks page miss, 3 clocks page hit, and 1 clock bursting. Write takes 5 clocks on a page miss, 3 clocks page hit.).

The performance boost a 485Turbocache Module will offer is 3 to 30 percent and is application sensitive.

6.8 485TURBOCACHE MODULE HARDWARE INTERFACE

The 485Turbocache Module has all of the control signals needed to interact with the i486 CPU. It decodes all bus cycles directly, and has the same timing and functionality for all i486 CPU pins. It is designed to operate without adding wait states during cache read hits and during transfers to the i486 CPU. The 485Turbocache Module has a bidirectional data bus for read and write accesses. The tag address bus contains the tag that is used for comparison with the previously stored tags.

The 485Turbocache Module supports burst operations similar to the processor's, transferring four doublewords to fill a line in the cache. As in the i486 processor, the 485Turbocache Module performs cache invalidation via a snoop address that resides on the address bus.

All signals and functions have been defined to allow the 485Turbocache Module to be designed as an optional add-in solution. A system can be designed with one or more sockets for one or more 485Turbocache Modules so the user may improve system performance as necessary by installing additional modules. The hardware can be designed so that it will automatically reconfigure the system when a 485Turbocache Module is added.

6.8.1 Pin Description

This section provides a summary of the signals and pins used in 485Turbocache Module. It is not a complete description, but provides an overview of function. See 485Turbocache Module data sheet, Order Number 240722 for a complete description. Note that pins using the same name as the i486 CPU are directly connected to the same i486 CPU pin.

6.8.2 Control Signals

Pin Name	Type	Description
ADS#	I	**ADDRESS STROBE** generated by the i486™ microprocessor. It is used to determine that a new cycle has been started.
M/IO#	I	**MEMORY/IO** cycle definition signal which used to indicate a Memory or IO access.
W/R#	I	**WRITE/READ** cycle definition signal which used to indicate a Write or Read access.
START#	O	**MEMORY START** is normally activated in the first T2 and indicates that a cache read miss or a write has occurred and that the current access must be serviced by the memory system. START# is not activated for IO cycles, and is not asserted if CS# is inactive. START# is always valid and must be logically ORed with other START# signals from additional modules.
BRDYO#	O	**BURST READY OUT** is a burst ready signal driven by the 485Turbocache Module to the i486 CPU. It is activated when a read hit occurs to the 485Turbocache Module.
CBRDY#	I	**CACHE BURST READY IN** is the burst ready input from the memory system. It is applied to both the 485Turbocache Module and the CPU in parallel. CBRDY# is ignored during T1 and idle cycles. BLAST# determines the length of the transfer. All cacheable read cycles are 4 dword transfers.
CRDY#	I	**CACHE READY IN** is the non-burst ready input from the system. Like CBRDY#, it applied to both the cache and CPU in parallel. CRDY# is ignored during T1 and idle cycles.
BLAST#	I	**BURST LAST** is output by the CPU and is sampled by the 485Turbocache Module to determine when the end of a burst transfer will occur.
BOFF#	I	**BACKOFF** is an input sampled by the i486CM to indicate that a cycle be immediately terminated. If BOFF# is sampled active, the 485Turbocache Module will float the data bus if it is currently active. The 485Turbocache Module will ignore all cycles, except invalidation cycles, until BOFF# is deactivated.

Pin Name	Type	Description
PRSN#	O	**PRESENCE** may be used as a 485Turbocache Module presence indicator. It should be connected via a 10K pullup resistor.
CLK	I	**CLOCK** is the timing reference from which the 485Turbocache Module monitors and generates events. CLK must be the same as the i486 CLK.
RESET	I	**RESET CACHE** forces the 485Turbocache Module to begin execution in a known state. It also causes all cache lines to be invalidated.

6.8.2.1 ADDRESS SIGNALS

Pin Name	Type	Description
A2-A31	I	**PROCESSOR ADDRESS LINES A2-A31** are the address lines used by the 485Turbocache Module. Address lines A2 and A3 are used as burst address bits. For 64K modules, A4-A14 comprise the set address inputs to the 485Turbocache Module and A15-A31 are used as the tag address. In 128K modules, A4 becomes a line select input, A5-A15 is the set address input and A16-A31 is used as the tag address.
BE0-BE3	I	**BYTE ENABLE** inputs are connected to the CPU byte enable outputs. They are specifically used for completing partial writes to the 485Turbocache Module in write hit cycles. They cause the cache to ignore read cycles.
CS#	I	**CHIP SELECT** is used to cascade 485Turbocache Modules. Address bits may be decoded in order to cascade multiple devices or be decoded to selectively cache portions of memory.

6.8.2.2 DATA SIGNALS

Pin Name	Type	Description
D0-D31	I/O	**PROCESSOR DATA LINES D0-D31** are connected to the system data bus. D0-D7 define the least significant byte while D24-D31 define the most significant byte.
DP0-DP3	I/O	**DATA PARITY** are the bits associated with the data on the data bus. Parity is treated by the 485Turbocache Module as additional data bits to be stored. Parity is important as commonly used 4-transistor SRAM cells are susceptible to soft errors.

6.8.2.3 CACHEABILITY SIGNALS

Pin Name	Type	Description
CKEN#	O	**CACHE ENABLE TO CPU** is the KEN# term generated by the 485Turbocache Module to the i486 microprocessor. CKEN# is activated twice; first during T1 to enable a cache line fill, and second before the last BRDY# to validate the line fill. CKEN# is ALWAYS active in T1, but will not validate a line fill if the line fill is a write protected line and WPSTRP# is low, or if the cycle is a read miss.
SKEN#	I	**SYSTEM CACHE ENABLE** is an input from the main memory system to indicate whether the current line fill is cacheable in the 485Turbocache Module.
FLUSH#	I	**FLUSH CACHE** causes the 485Turbocache Module to invalidate its entire cache contents regardless of CS#. Any line fill in progress will continue, but will be invalidated immediately. The i486 CPU flush instruction does not affect the 485Turbocache Module.
WP	I	**WRITE PROTECT** defines a line as write protected. WP is maintained internally as a state bit. Any subsequent writes to a write protected line will have no effect. In 128K configurations where the 485Turbocache Module is configured as 2 lines per sector, write protection is defined on a sector, not line, basis.
WPSTRP#	I	**WRITE PROTECT STRAPPING OPTION** changes the behavior of CKEN#. CKEN# is always asserted in T1 to indicate a cacheable line transfer. CKEN# is also asserted the clock before the last transfer of a line fill from 485Turbocache Module to i486 CPU. If WPSTRP# is strapped low, and a write protected line is being transferred, CKEN# is not activated before the last transfer.

6.8.2.4 SNOOP SIGNALS

Pin Name	Type	Description
EADS#	I	**VALID EXTERNAL ADDRESS STROBE** indicates that an invalidation address is present on the address bus. The 485Turbocache Module will invalidate this address, if present, but will only do so if CS# is active. The 485Turbocache Module is capable of accepting an EADS# every other clock.

6.8.3 System Configuration and Processor Interface

This section discusses the 485Turbocache Module operation in relation to two interfaces: the CPU-to-485Turbocache Module interface, and the main memory and bus controller-to-485Turbocache Module interface.

6.8.3.1 i486 CPU CONNECTIONS AND TAG MAPPING

When single or multiple 485Turbocache Module devices are connected to an i486 processor system, the processor's internal cache should map the entire address space including that of the 485Turbocache Module devices to provide the highest performance. This is the most efficient configuration. The i486 CPU can complete internal cache read hits in a single clock cycle, and the 485Turbocache Module provides the next fastest access in two clocks for the first doubleword and the remaining three doublewords in three clocks.

On reads, when the i486 processor cache has an internal cache hit, ADS# is not asserted, and the 485Turbocache Module does not begin a new cycle. Otherwise, ADS# is asserted and data is accessed from the 485Turbocache Module, from main memory, or from an I/O device.

No matter how many 128-Kbyte caches are cascaded, the set and tag addresses are connected to the same pins on the 485Turbocache Module. The processor's address bits A2–A31 are connected to A2–A31 on the 485Turbocache Module. Internally, address bits A4–A15 are sent to both sets, to select one of 4,096 locations. Because the cache is two-way set associative, each address points to information stored in two banks. On each read or write cycle, the value of A16–A31 is compared to the tags stored at the location addressed by A4–A15. If they are equal, and if the valid bit is set, then a hit occurs. If a read cycle is in progress, then the 485Turbocache Module returns data to the i486 CPU. If the hit cycle is a write cycle, then the new data is updated in the 485Turbocache Module.

When multiple 485Turbocache Modules are used the chip select starts by decoding A16 onwards. For example, with a 256-Kbyte cache A16 and A17 are decoded for generating the CS#. The set and tag addresses of a system with four 485Turbocache Modules used as shown in Figure 6-15.

The BRDYO# output and the CBRDY# input must be used in forming of the i486 CPU's BRDY# input. Similarly, the CRDY# input must be used in forming of the i486 CPU's RDY# input. Signals that are common to the i486 CPU and the 485Turbocache Module include BOFF#, BLAST#, EADS#, BE0#-BE3#, and DP0-DP3.

The memory system generates KEN# to the i486 CPU when read data needs to be cached. The 485Turbocache Module receives this signal as the SKEN# input and produces CKEN# when appropriate. The 485Turbocache Module's CKEN# output can be used in the formation of the KEN# input to the i486 CPU. CKEN# can be used in conjunction with other logic that can deassert KEN# to the CPU when the system wants the current line fill to be cached by the 485Turbocache Module and not cached in the i486 CPU. The CKEN# signal is always asserted in T1, but is then deasserted if CS# is inactive.

The 485Turbocache Module connects directly to the i486 CPU's address lines A2-A31. The designer may have to add external buffers to the address outputs, depending upon the loading. Other signals connected to the i486 CPU include the burst control signals, the bus cycle definition signals, the byte enables, the ADS# signal, and the data and

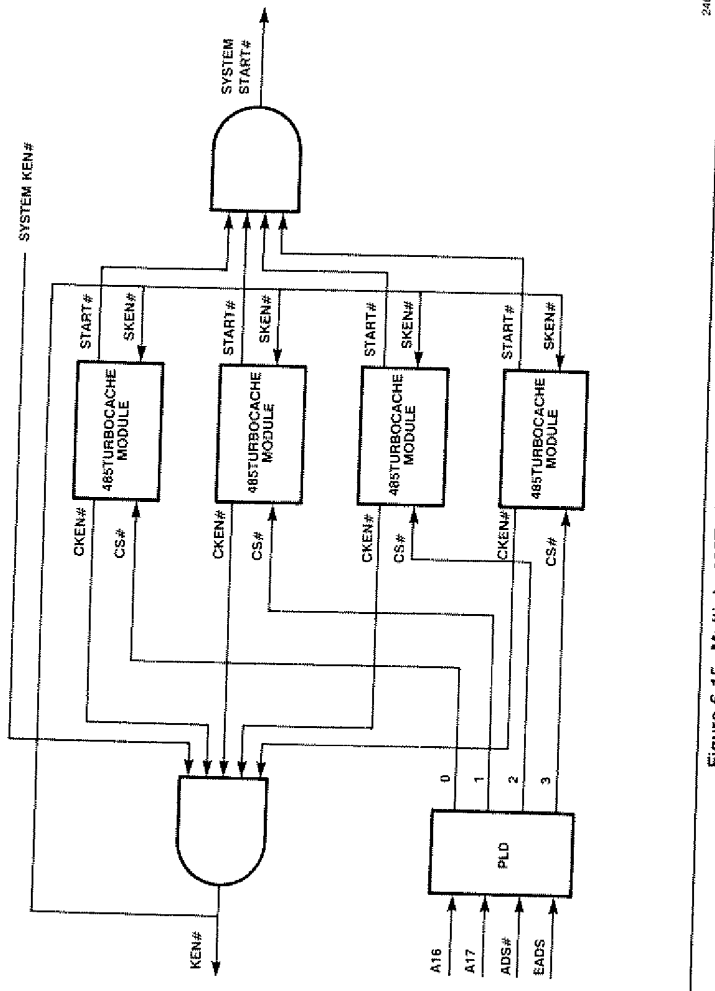


Figure 6-15. Multiple 485Turbocache Module Configuration

parity signals. The 485Turbocache Module and CPU connections are shown in Figure 6-16. The 485Turbocache Module, main memory controller, and bus controller interface are shown in Figure 6-17.

6.8.3.2 READ HIT CYCLES

A read hit cycle occurs when requested data is present in the 485Turbocache Module. The i486 CPU attempts to retrieve the entire line from the 485Turbocache Module without incurring wait states. This may be accomplished by activating the KEN# input at the end of T1 (the clock in which ADS# becomes active). There is very little time to decode the address, generate the KEN# signal to the i486 CPU, and complete a zero wait state read operation. Because KEN# is sampled twice, it is possible to always assert KEN# in T1 and to wait until the end of a line fill to decide whether the data is cacheable.

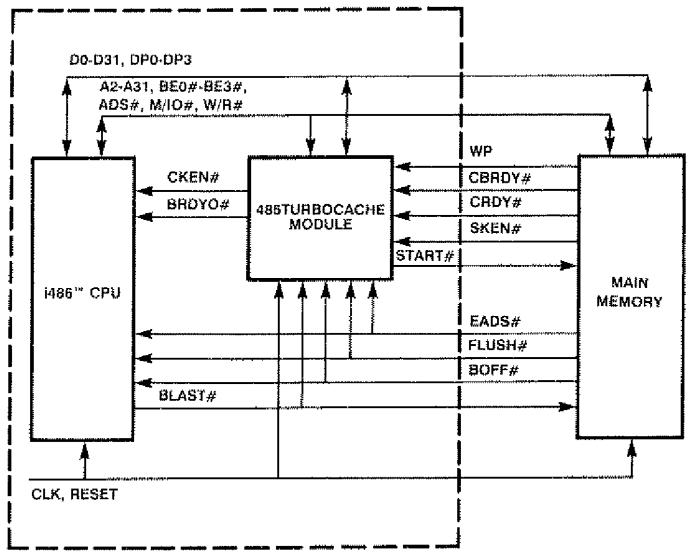


Figure 6-16. 485Turbocache Module and i486™ CPU Connections

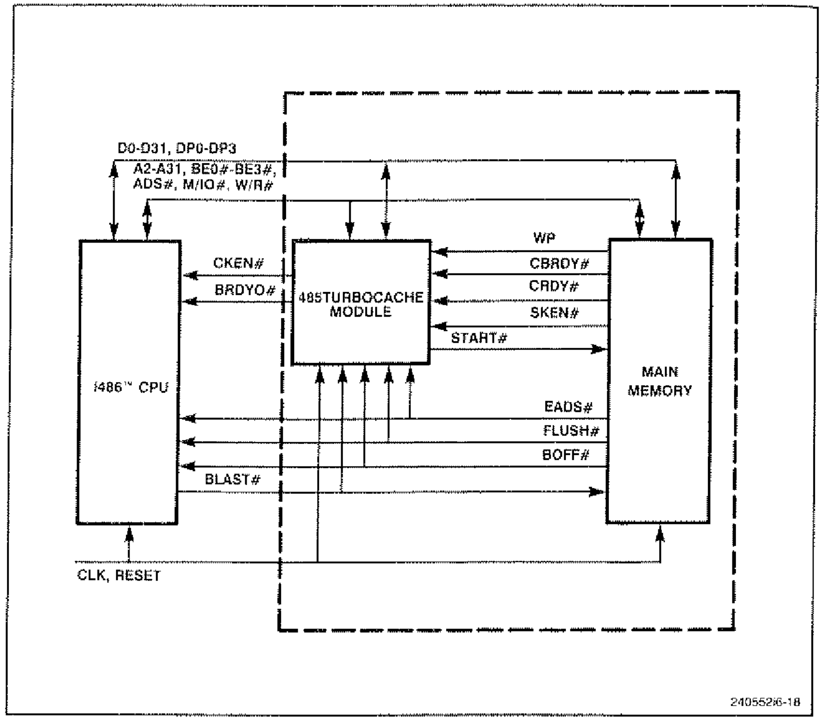


Figure 6-17. 485Turbocache Module and Main Memory Connections

CKEN# is used in the formation of the KEN# signal to the i486 CPU. Therefore, CKEN# is always activated in T1 (see Figure 6-18, and Figure 6-19 in Section 6.8). If a read hit occurs, data can be sent to the i486 CPU in zero wait states and can still be cacheable to the processor's on-chip cache. The 485Turbocache Module asserts CKEN# which remains asserted for the duration of the read hit cycle (unless WPSTRP# is low and the line is write protected). This means that the i486 CPU will cache the entire line unless external logic is added to cause the KEN# signal to be sampled high in the clock before the last BRDYO# from the 485Turbocache Module.

If the CKEN# input from the 485Turbocache Module is connected directly to the KEN# input of the i486 CPU, then the CPU will always sample KEN# active at the end of T1. To deassert KEN# to the processor, the system must create another signal that is used in the formation of the i486 CPU's KEN#, and the 485Turbocache Module's SKEN#. Using this technique a non-cacheable, non-burst cycle can be performed.

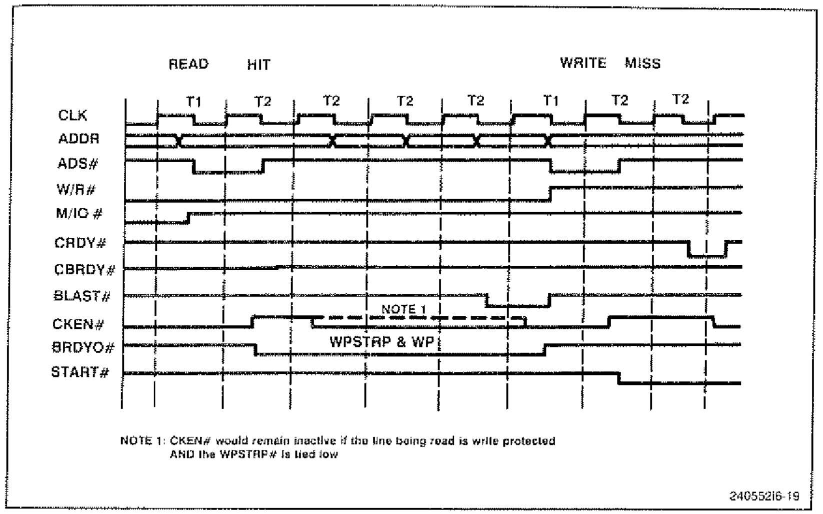


Figure 6-18. Read Hit—Write

The BRDY# signal to the i486 CPU can be generated from many sources. Therefore, the various signals should be logically "ORed" to generate the actual i486 BRDY# input.

On a cache read hit, the 485Turbocache Module generates a BRDYO# signal for each of the doublewords it transfers. The 485Turbocache Module asserts BRDYO# in the first T2 cycle, and BRDYO# remains asserted for the duration of the burst. If the i486 CPU either terminates a burst early or fails to generate a burst cycle as defined by BLAST#, the 485Turbocache Module will deassert BRDYO# after the i486 CPU has sampled the required data.

6.8.3.3 READ MISS CYCLES

On a cache read miss, the 485Turbocache Module initiates a system action, by asserting the START# signal. The system is responsible for generating a BRDY# or a RDY# signal to the i486 CPU. The 485Turbocache Module monitors the ready signals CRDY# and CBRDY# to determine when valid data appears on its data lines. If the system indicates that the cycle is cacheable, then the data is cached in the 485Turbocache Module. The system must transfer a complete line for the location to be cacheable. (See Figure 6-20 in Section 6.8.)

For read misses, CKEN# is asserted at the end of T1 but is deasserted during the first T2 of a read miss, and will remain inactive until the cycle is complete. The system may return a ready without ever activating the KEN# and SKEN# lines for a non-cacheable operation.

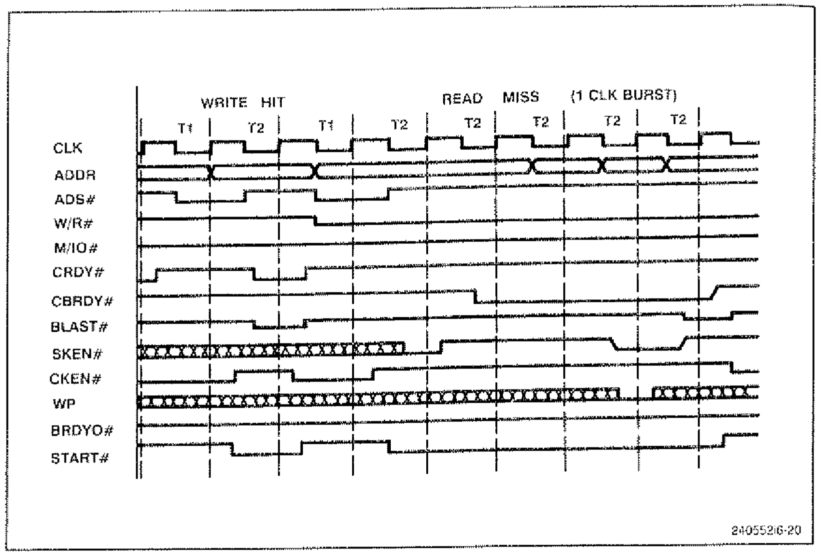


Figure 6-19. Write—Read Miss (Fastest Line Fill)

6.8.3.4 WRITE CYCLES AND I/O CYCLES

The 485Turbocache Module is a write-through cache, so main memory is updated with every write hit or miss. The 485Turbocache Module is not required to generate a ready signal to the i486 CPU for write cycles. However, it does perform a comparison and updates the cache memory when a write hit occurs (provided the location isn't write protected). The 485Turbocache Module is not updated on write misses. The timing diagrams for write operations are shown in Figure 6-18 and Figure 6-19 in Section 6.8.

The 485Turbocache Module ignores all I/O cycles. When an I/O cycle is executed by the i486 processor, the system responds and terminates the cycle. The 485Turbocache Module does not assert the START# signal for I/O accesses, and the system should monitor the M/IO# signal rather than wait for the assertion of the START# signal. The timings for an I/O read are shown in Figure 6-24 in Section 6.8.

6.8.4 System Interface

This section describes how general signals are used to control key actions of the 485Turbocache Module. The system design must generate or use these signals to efficiently use the 485Turbocache Module.

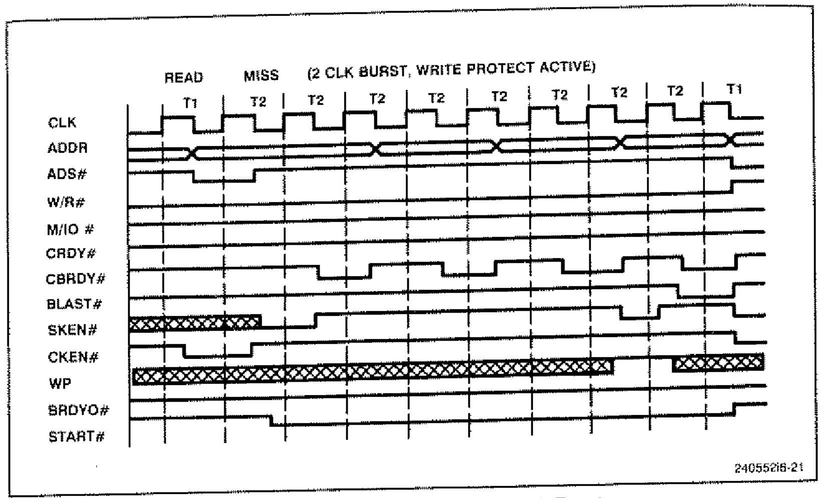


Figure 6-20. Read Miss—Two Clock Burst

6.8.4.1 READ MISS CYCLES

The system memory must return data to the processor for all cache read misses. The 485Turbocache Module asserts the START# signal on a read miss, and the START# signal indicates to main memory to begin a memory cycle. START# may be used to enable the data transceiver to route main memory data to the CPU.

6.8.4.2 LINE FILL

Because the 485Turbocache Module is a passive device, the i486 CPU must generate all of the control signals required in a line fill except for the START# signal. The CPU's KEN# input and the 485Turbocache Module's SKEN# input are asserted before the first ready is returned to the system. Once the line fill is initiated, the system can cache the line exclusively in the 485Turbocache Module by having SKEN# asserted and having the CPU's KEN# deasserted one clock prior to the ready of the last data word. The 485Turbocache Module will not cache data from a read miss when the first word of the access is returned in zero wait states. At least one wait state is required on the first doubleword transfer from the system to perform a cache update. The timings for a line fill are shown in Figure 6-19 in Section 6.8.

The order in which the doublewords are transferred is defined by the i486 CPU and is shown in Table 6-2. For example, if the first address was 104, then the next three addresses will be 100, 10C, and 108.

Table 6-2. Burst Address Sequencing Issued by the i486™ CPU

First Address	Second Address	Third Address	Fourth Address
0	4	8	C
4	0	C	8
8	C	0	4
C	8	4	0

NOTE: The address bits are the least significant bits.

The 485Turbocache Module performs its data updates in parallel with the i486 processor. The 485Turbocache Module updates its memory while the data and ready signals are returned by the system. The 485Turbocache Module does not penalize the i486 CPU's performance. Also, the 485Turbocache Module will only cache complete, 4 doubleword line transfers.

6.8.4.3 WRITE CYCLES AND WRITE PROTECTION

Because the 485Turbocache Module is a write-through cache, writes are immediately forwarded to the system. If a processor write occurs on a valid entry that is not write protected, the new data will be stored into the memory in zero wait states. The 485Turbocache Module will not generate a ready signal. It is the system's responsibility to update the system memory on all writes and to terminate all cycles with a ready signal. Even after the 485Turbocache Module has completed its internal write update, it remains idle until the system returns a ready to the processor.

A cache location can be write protected by asserting the WP input to the 485Turbocache Module. The WP signal must be valid during the third BRDY# or RDY# of a cache line fill cycle. It sets a state bit within a particular cache location and remains in effect until the bit is invalidated. Tieing WPSTRP# low will not allow the write protected entry to be cached by the i486 CPU in subsequent accesses. The entry can be invalidated by any of the following: a flush operation, a reset operation, an invalidation cycle, or an LRU replacement.

When an i486 CPU cycle produces a write hit to a write-protected 485Turbocache Module location, data in the cache is not modified. The 485Turbocache Module responds in the same way whether or not a write hit location is write protected by asserting the START# signal. It is the designer's responsibility to prevent inconsistencies between the 485Turbocache Module and main memory when using the WP signal.

6.8.4.4 SYSTEM CACHEABILITY INDICATION

The 485Turbocache Module uses the cache enable scheme of the i486 CPU. A cache update to the 485Turbocache Module requires activating the SKEN# signal. The signal is sampled twice, first with or after START# and before the first ready signal from

BRDY# or RDY#, and again on the rising clock edge before the last ready. If SKEN# was deasserted at either of the specified sample times, then the access is considered non-cacheable. SKEN# is ignored during write cycles.

Typically, the system will use the same logic to generate the i486 CPU's KEN# signal and the 485Turbocache Module's SKEN# signal. However, it is not necessary for both have to be asserted during an access. It is possible to use different cacheing maps for the CPU cache and the 485Turbocache Module cache because the i486 CPU and the 485Turbocache Module maintain their own cache contents via snooping.

6.8.4.5 INVALIDATE CYCLES

The 485Turbocache Module has the same snooping mechanism as the i486 CPU. Since the devices operate in parallel, they can snoop in parallel. Because both of the devices are write-through caches, it is only necessary for them to detect cache write cycles that are initiated by other bus masters.

EADS# indicates to the 485Turbocache Module that the address pins contain an address for invalidation. This address is normally generated by an external master but if AHOLD is inactive, then a snoop comparison is initiated on the address driven by the i486 CPU. EADS# is recognized regardless of the AHOLD and BOFF# signals. The fastest invalidation rate for the 485Turbocache Module is one invalidation on every clock cycle, whereas the i486 CPU can support snoop addresses on every clock cycle. CS# must be asserted for the 485Turbocache Module to recognize invalidation cycles.

6.9 DESIGN CONSIDERATIONS

This section deals with the main memory interface, cascadable caches, and multiple CPUs. Any design using the 485Turbocache Module must consider these issues to use it effectively.

6.9.1 DRAM Interface

In most applications, the 485Turbocache Module interfaces to a DRAM-based memory system which is the main memory. The START# signal can accomplish this interface. START# is asserted for all memory write cycles and read miss cycles and indicates that memory must process the current cycle.

The START# signal can be used to control the data transceivers from the memory system. During a read cycle, the memory controller determines that a page hit has occurred and initiates a memory access with a CAS cycle. If the 485Turbocache Module determines that the current cycle is a miss, then the START# signal is asserted, the data transceivers are enabled, and memory data is transmitted from memory to the i486 CPU. If the cycle was a read hit, then the data transceivers are disabled. During write operations the memory system ignores the START# signal.

The START# signal can also be used as an ADS# signal for the main memory. In this configuration, a memory cycle cannot begin until START# is sampled active. This provides maximum memory availability to DMA and other memory devices. Since the START# signal is not sampled until the end of T2, at least an extra wait state is added for main memory accesses.

Note that although START# normally appears in the first T2 of a memory cycle, it may not if the previous cycle contained an invalidation request.

If the 485Turbocache Module has been installed as a system option and is not present, the START# signal will never be asserted. A memory controller must take this into account and automatically begin a cycle if PRSN# (presence indicator) is sampled inactive.

6.9.2 Cascadable Cache

The 485Turbocache Module can be cascaded to configure a deeper cache memory for the processor. Up to four can be used to provide as much as 512 Kbyte of cache. This section discusses such designs.

6.9.2.1 SYSTEM CONTROL SIGNALS AND CASCADABLE CACHES

The START# signal used by memory is the logical OR for each individual 485Turbocache Module START# output. If any cache has information that is needed by the processor, then its START# signal is at a high level, and it inhibits the main memory START# signal (as there is no need to access the main memory). If needed data is not present in any of the 485Turbocache Modules, then the START# signals are low, and main memory data is accessed.

The KEN# input to the i486 processor should be a logical OR for each of the 485Turbocache Modules and for a memory controller output. The memory controller output can be asserted high to indicate that the information to the i486 CPU is non-cacheable.

The SKEN# signal is the cache input to the 485Turbocache Module. The memory controllers must assert SKEN# when a transfer to the 485Turbocache Module is cacheable. The SKEN# inputs for all of the 485Turbocache Modules must be tied together. The controller that has its CS# asserted determines which cache will receive the information.

The EADS# signal from the memory controller must be connected to the i486 CPU and to all of the 485Turbocache Modules. In this way, invalidation cycles are executed in all the 485Turbocache Module devices simultaneously.

This section reviews cascaded 485Turbocache Module configurations (see Section 6.6.2.1). The entire memory space is covered in a single cache or a cascaded cache configuration. When multiple 485Turbocache Modules are used, only one 485Turbocache Module is selected by asserting the CS# pin.

The tag address connections are shown earlier in Figure 6-15 for a 512-Kbyte cache. For example, TA0 through TA15 are always connected to A16 to A31. In the configuration with one 485Turbocache Module, the chip select is grounded. In the two 485Turbocache Module configuration, A16 is used to decode between the two caches. In the four 485Turbocache Module configuration, A16 and A17 are used to generate the CS# signals. These configurations are summarized in Table 6-3.

6.10 TIMING DIAGRAMS

This section shows the 485Turbocache Module interface signals for the standard execution cycles.

6.10.1 Read Hit Followed by a Write Miss

Figure 6-18 shows a read hit cycle followed by a write miss cycle. During the read hit, BRDYO# is asserted in the T2 cycle, and remains asserted until the entire line has been transferred to the i486 CPU. The CKEN# signal is asserted at the end of T1 to begin a line fill and is asserted throughout the transfer. CKEN# is deasserted in T2, and will remain deasserted only if a write-protected line is being transferred and WPSRTP# is low. START# is not asserted for a read hit but is asserted for the write cycle.

Write hits and write misses are indistinguishable to the system. On a write hit, the 485Turbocache Module is updated in zero wait states. The system is responsible for terminating all write cycles.

6.10.2 Write Hit Followed by a Read Miss

Figure 6-19 shows a write hit followed by a read miss. The read miss operation shown is the optimal line fill for the 485Turbocache Module. The START# signal is asserted for the write in the first T2. Once it is determined that the cycle is a read miss, START# is asserted again to request a main memory access. CKEN# is asserted in T1, but is deasserted once a miss has been determined. The system monitors SKEN# to determine whether the current access is cacheable. Because SKEN# is asserted by the system

Table 6-3. Tag Address and Address Connections

Cache Size in Bytes	Tag Address TA0-TA15	CS#
128K	A16-A31	Ground
256K	A16-A31	A16/$\overline{\text{A16}}$
512K	A16-A31	A16, A17 decoded

during both sample windows, the line is allowed to be stored in the 485Turbocache Module data RAM as a valid entry. Note that the first read of a 485Turbocache Module burst requires at least one wait state.

In this example, WP is sampled during the line fill. WP is low during the sample window, indicating that the line is not write protected.

6.10.3 Read Miss with a 2-Clock Burst Transfer and Write Protection

Figure 6-20 shows a read miss which has a 2-clock burst transfer rate. There are two cycles between each of the CBRDY# signals that are returned to the 485Turbocache Module. These indicate when data is valid for storage in the 485Turbocache Module. The line is cacheable because SKEN# is asserted at the appropriate sampling time.

In this example, set-up and hold times of the WP signal are timed to the clock edge before the last CRDY# or CBRDY#.

6.10.4 2-Clock Snoop

Figure 6-21 shows a 2-clock snoop to the 485Turbocache Module. The timings show that the optimal invalidation rate equals the rate at which EADS# can be received by the 485Turbocache Module. In this case, the maximum invalidation rate is one EADS# on every other clock cycle. Note that if the 485Turbocache Module is transferring data during the invalidation, the transfer will continue but that transferred line will also be invalidated.

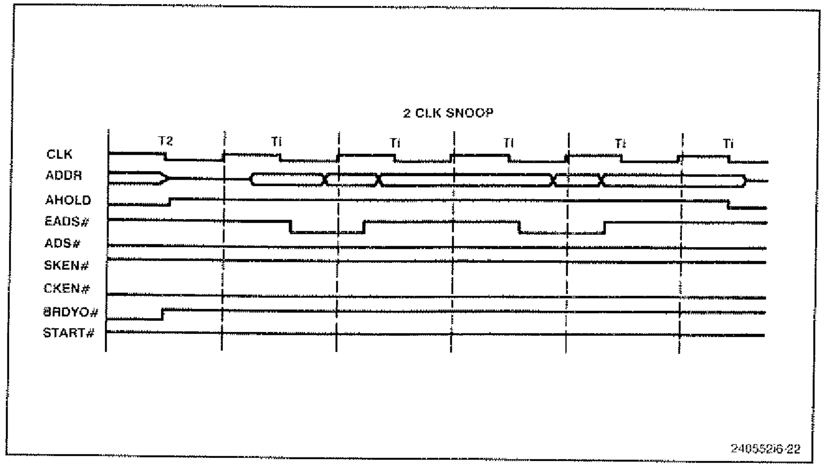


Figure 6-21. Snoop Cycle—Maximum Invalidation Rate

6.10.5 Snoop Cycle and Read Hits

Figure 6-22 shows a snoop cycle occurring in parallel with a 485Turbocache Module read hit. The 485Turbocache Module is able to continue data transfers to the i486 CPU while the snoop is being executed. The designer must observe the setup and hold time requirements for the snoop address. This line will be invalidated as well as the line at the invalidation address.

6.10.6 Backoff Cycles during a Read Hit

Figure 6-23 shows a read hit cycle in progress with a 485Turbocache Module. The system asserts the BOFF# signal. In response, the read cycle must be aborted and the address, control, and data buses have to be in high impedance for the next clock cycle. Only the data bus is active in the 485Turbocache Module. Figure 6-23 shows that in response to BOFF# the 485Turbocache Module will float the data bus during a read hit in the next clock cycle.

6.10.7 I/O Cycle Followed by a Read Miss

Figure 6-24 shows an I/O cycle followed by a read miss. The 485Turbocache Module does not assert START# for I/O cycles. It is the designer's responsibility to decode the cycle as an I/O cycle, and intiate an I/O cycle using external logic.

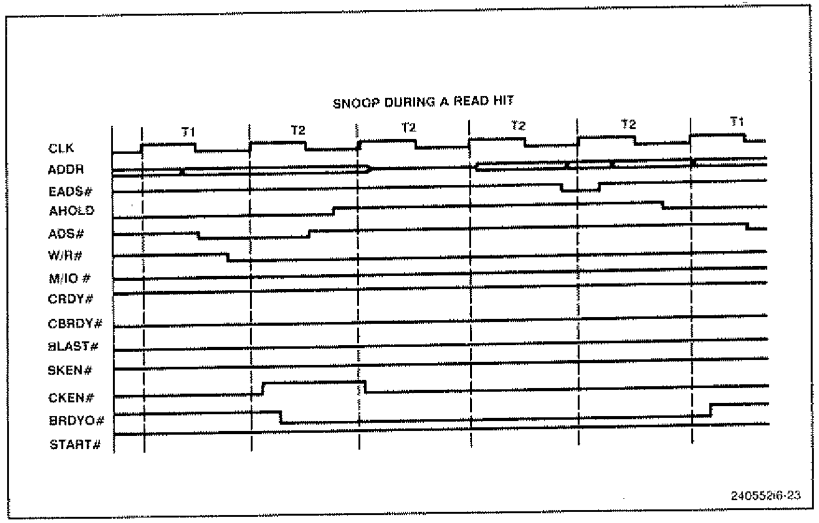


Figure 6-22. Snoop during Read Hit

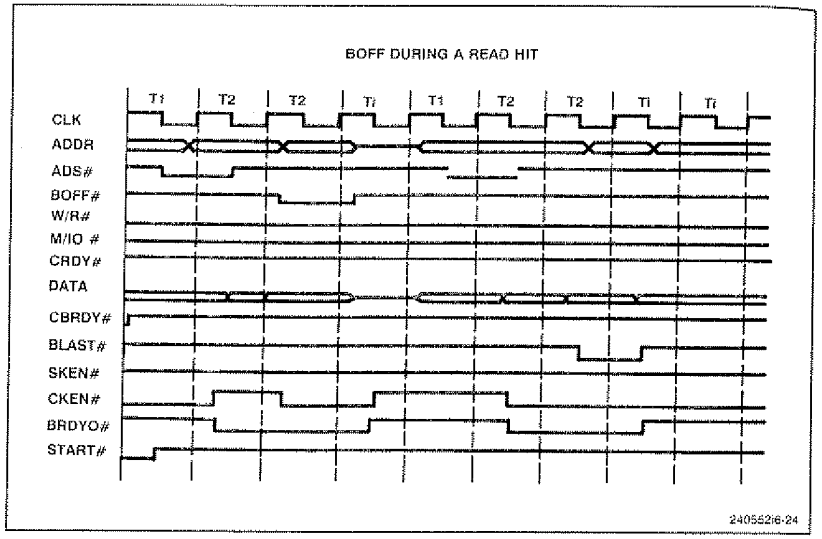


Figure 6-23. BOFF# during Read Hit

Figure 6-24 also shows a read miss with an interrupted burst. As shown, the 485Turbocache Module supports line fills which are burst, non-burst or a combination of the two. START# remains asserted throughout the complete line fill. All other address and control signals are driven by the i486 CPU.

6.10.8 Non-Cacheable Read Miss

Figure 6-25 shows a read miss cycle that the system has defined as a non-cacheable, single transfer. CKEN# will cause the i486 CPU's KEN# input to be asserted at the end of T1. To make the accessed data non-cacheable, the system must return ready without activating the i486 CPU's KEN# signal or 485Turbocache Module's SKEN# signal. This transfer will be a single cycle, non-cacheable read by the i486 CPU and the 485Turbocache Module.

The latter part of the figure shows a read hit cycle in which a doubleword is read.

6.11 SUMMARY

External caches can help i486 CPU performance or reduce main memory costs. For systems with multiple CPU's, they are required to reduce main memory traffic and avoid main memory contention. Single CPU systems, at 25 and 33 MHz, work best with an optional cache. The 485Turbocache Module is ideally suited as an option for systems with a good main memory design.

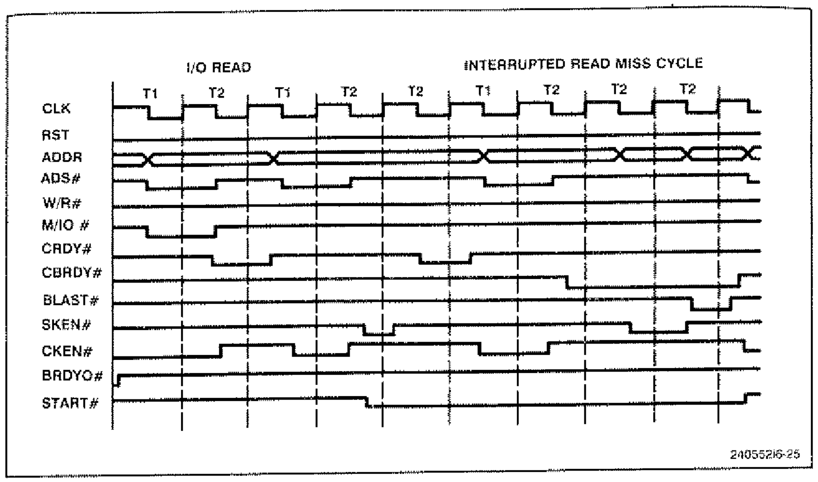


Figure 6-24. I/O Cycle—Interrupted Burst

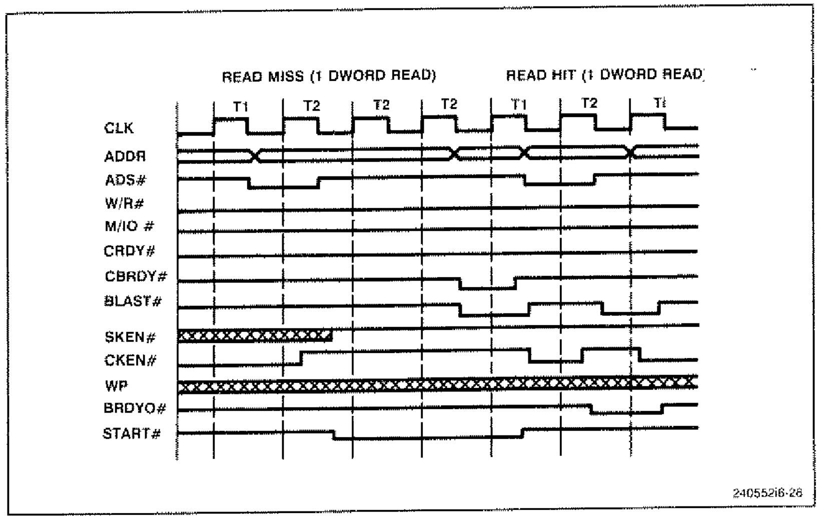


Figure 6-25. Single Cycle, Non-Cacheable Read

EXHIBIT 4

DOCKET M_1018 US

APPENDIX (PART 2)

MOSEL'S BURSTSRAM ARCHITECTURE SPECIFICATION

BY

ALFRED K CHAN

REV 1.3

7 February 90

Table of Contents

Note: The author wants to thank Jason Lin for his effort in generating the timing diagrams.

1.0 Structural Description

1.1 Array organization

Mosel **Burst-ram** can interface to both **80386** and **80486** systems synchronously. It contains a 2 by 2k x 36 array organization (see figure 6). To support a 32 bit system, four **Mosel burst-rams** will offer the minimal 64K byte 2 way associative organization or 64 K byte direct mapping organization. Each **burst-ram** contains two nine bits wide input/output data ports. Each data port contains eight bits of data and one bit of parity. More **burst-rams** can be used for larger cache size.

Each **Mosel's burst-ram** contains two banks of 2k x 36 array. Each array is subdivided into four subarrays. For burst-rams positioned at the most significant byte of the local 32 bit bus, **sub-array A** (2k x 9) will contain the most significant data byte and the corresponding data parity bit of the fourth word (highest address) of a quad fetch. Similarly, **sub-array B** (2k x 9) will contain the most significant data byte and the corresponding data parity bit of the third word and **sub-array C** will contain the most significant data byte and the corresponding data parity bit of the second word.

1.2 Mosel Burst-ram address path

Mosel's burst-ram contains two address registers. **HADDR** contains the hit address and **MADDR** register contains the miss address or replace address. Address from the host will be latched into **HADDR** register in the burst-ram through activation of **HALE**. A high level on **HALE** will open **HADDR** latch to the address decoder of data array in the burst-ram. **HALE** will be activated the same **CLK** clock as **ADS#**. **HADDR** address will be latched into **MADDR** register through the activation of **MALE**. **MALE** will be activated one clock after **ADS#** assertion if a miss

MOSEL CONFIDENTIAL

is detected. The activation of **QWR** will always direct **MADDR** register content to the address decoder of the data array in the burst-ram.

1.3 Mosel Burst-ram datapath

Mosel's burst-ram is dual ported. It allows read/write hit operations to run in parallel with miss processing. It decouples the main memory subsystem from the host local bus through an interface that uses internal temporary holding registers. This allows an easy migration path for system design from **386** to **486** without significant redesign of the main memory subsystem. Upgrading from **386** system to **486** purely become a local bus issue and is supported by the **Mosel Cache Controller**.

The host port supports both scalar reads and writes in **386** systems. For **486** systems, in addition to scalar reads and writes, **Mosel burst-ram** will support burst read at the host port. For scalar read and write, the burst-ram appears to the host port just like any standard sram. The addition of temporary registers does not induce any additional delay to the scalar read/write path. As a result, the access time for scalar reads will be the same as standard cache data rams.

Mosel's burst-ram architecture makes burst-mode support easy. It contains a total of three 36 bits temporary holding registers (see figure 6). **MRHREG** is used to support local host bus (**486**) burst read. A burst read hit will cause all four 32 bit data doublewords within the same line and four data parity bits to be read into **MRHREG**. **"Wrapped-around"** burst order is supported. The first demand word will be fetched and sent to the host port directly through the bypass at **BMUX**. The subsequent three remaining words are now fetched from **MRHREG** instead of through the data array. This architecture allows very high speed (**50Mhz**) burst mode possible without using ultra-fast (**sub 10ns**) data rams. Burst data from **MRHREG** now only need to drive the data output buffers. Burst-order is controlled by **HA<3:2>** and is totally transparent to the burst-ram.

For miss operations in **write-back** mode, **MWBREG** holds data from selected data line to be replaced and write back to main memory if the replace line contains **valid** and **dirty** data. This allows burst write on the system memory port **SP<8:0>** without binding the data array for write back replacement cycles. This arrangement allows the cache data array to serve the host port for local bus read and write hits. For slow main memory subsystems with large number of wait states,line replacement cycles could be long.

The **MUPREG** contains quad fetch miss data from main memory. The **MUPREG** contains the entire line of quad fetch data. The order of loading the **MUPREG** register is controlled through **SA<3:2>** making burst-order to main memory purely transparent to the burst-ram. Once the line is loaded into **MUPREG**, the entire line can be loaded into the cache data array by activating **QWR** signal.

Each **Mosel burst-ram** offers 36 bit data (4 data parity) read and write between the data array and the holding registers. Four **Mosel's burst-rams** in a 32 bit system will offer a true line (128 bits) read and write in one clock. Subsequent access of a burst read through holding registers drastically reduce burst data access time.

1.4 Mosel Burst-ram Bank Select

Each **Mosel burst-ram** contains 2 banks of 2k x 36 data array.

With respect to the host port **HP<8:0>**, a low assertion of **HW0#** signifies a write request to bank0 through **MWREG** and a low assertion of **HW1#** signifies a write request to bank one through **MWREG**. The inactivation of both **HW0#** and **HW1#** implies a read operation to the data array. **G0#** and **G1#** are active low output enables. A low assertion of **G0#** will gate the read data from bank0 to host port and the low assertion of **G1#** will gate the read data from bank one to the host port. **HW0#**, **HW1#**, **G0#** and **G1#** cannot be active simultaneously.

With respect to the system port **SP<8:0>**, a miss detection, as indicated by a **MALE** (**M**iss **A**ddress **L**atch **E**nable) assertion, will trigger the latching of bank select information derived from **G0#**, **G1#**, **HW0#** and **HW1#**. This bank select information is remembered by the burst-ram so that miss data fetched from main memory can be updated to the correct data bank. This information will be remembered until a **QWR** (quad write) activation or a **RESET** assertion. Similarily, the **SELECT#** (burst-ram chip select) signal is latched through **MALE** for the system port miss processing operations between the burst-rams and main memory.

For systems that support **write-through** cycles, (refer to section 2.2.5) a **BYPASS** signal activation together with **SW#** low assertion will cause both host and system port of the corresponding burst-ram bank to be selected. (see figure 7,8) The **BYPASS** high and **SW#** low assertion will over-ride and reset the **SELECT#** information latched through **MALE** activation. Cache controller controlling the burst-ram will have to make sure that all miss processing operations are completed at system port before any "**bypass**" **write-through** write cycles are triggered.

2.0 Functional Description

2.1 386 Burst-ram operations

2.1.1 386 Scalar Read and Write Hits

Mosel's Burst-ram allows scalar read/write operations with local bus processor through the host port **HP<8:0>** (see figures 1 and 2) The host port interface with **386** just like any regular standard by eight srams. If data parity is not checked as in most **386** systems, **HP<8>** and **SP<8>** can be left disconnected. Scalar read hits will require two bus states to complete. **HALE** will latch the address into **HADDR** register. At the same time, data array in the burst-ram are accessed and read to the local processor through the host port.

For systems already using **Intel 80385**, users of **Mosel Cache Controller** may use the existing external data transcievers to latch write data if desired. However, **MWREG** may be used as the buffered write latch. A write hit will always results in data from **MWREG** to be written into the data array.

2.1.2 386 Write Miss Operations

2.1.2.1 386 Write Miss Operations in Write-though mode

A write miss will result in **MALE** activation in the next clock after **ADS#**. **MALE** activation will inhibit **MWREG** data from being written into the data array in the current clock. In **386 write-through** mode, data **MWREG** will be sent to system port **SP<8:0>** by activating **BYPASS** and **SW#** signals. Hence, a true write into the data array will occur only if **MALE** is inactive and **HW0#** or **HW1#** is asserted low. A write miss operation in **write-through** mode will require only two bus states. In another words, zero wait state buffered writes is supported for write miss.

2.1.2.2 386 Write Miss Operations in Write-back mode

In **386 write-back** mode with write miss, **HW0#,HW1#** will be asserted low for one additional clock after **MALE** is deasserted to allow write miss data from **MWREG** to be written into the data array .
A write miss operation with burst-ram in **write-back** mode will require a minimum of three bus states. At the end of the second bus state, the rising edge of **BMUXC<0>** will trigger the latching of the data from the selected replace line into **MWBREG**. A **write-back** cycle of replace data will occur if the replace data is **valid** and **dirty**.

MUPREG will be used to hold quad fetch data from main memory. Miss data will be fetched in a "**wrapped-around**" fashion with the demand word fetch first. Each byte of the **MUPREG** is associated with a valid bit. As each 36 bit data (4 data parity bits) is updated into **MUPREG** through **SRDY#** assertion, the corresponding valid bit will be set if **QWRWQ** is active. The inactivation of **QWRWQ** signal inhibits **dirty** and **valid** miss data of the same line from being overwritten by quad fetch old data. This **valid** bit will qualify the corresponding byte of the **MUPREG** to be written into the burst-ram data array as **QWR** is asserted. After each **QWR** (quad write cycle) all valid and mask bits associated with **MUPREG** will be reset.

Mosel's **burst-ram** supports "**advance**" data array update for write misses in **write-back** mode. "**Advance**" write miss processing implies that write miss data can be updated into data array in burst-ram without waiting for the miss line to be fetched from main memory. Associated with each byte of the **MUPREG** is a "**mask**" bit. The corresponding "**mask**" bit will be set as an "**advance**" write occurs for the write miss data from

MWREG. The setting of the **mask** bit will inhibit the corresponding data byte in **MUPREG** from updating the data array inside the burst-ram.

2.1.3 <u>386 Read Miss Operations</u>

A miss detection will result in **MALE** activation the following clock after **ADS#** assertion. Miss address will be latched into **MADDR** register. **MUPREG** will be used to hold quad fetch data on read miss from the main memory. Miss data will be fetched in a **wrapped-around** fashion with the demand word fetched first. The demand word will be sent directly to the host port through activation of **BYPASS** signal. **386** host now satisfied with the demand data can continue its processing while the remaining three words within the same miss line are fetched from main memory. Meanwhile, other read hits can be processed in parallel on the host port with the burst-ram data array directly accessible by the host. Since the cache is going to have above 90% hit rates, it is likely that the next bus request after a read miss is a hit. In other words, there is a very high chance to attain parallelism between hit and miss processing.

Each byte of the **MUPREG** is associated with a **valid** bit. As each 36 bit data word (4 data parity bits) is updated into **MUPREG** through **SRDY#** assertion, the corresponding **valid** bit will be set if **QWRWQ** is active. The inactivation of **QWRWQ** signal inhibits dirty and valid miss data of the same line from being written by quad fetch miss data. This **valid** bit will qualify the corresponding byte of the **MUPREG** to be written into the burst-ram data array when **QWR** is asserted. After each **QWR** (quad write cycle), all **valid** and **mask** bits associated with **MUPREG** will be reset. The presence of **valid** bits allow abortion of quad fetch on read miss for a subsequent pending write request in **write-through** mode to access the main memory interface. **Valid** bits make partial write of a quad fetch line possible.

Interface of the burst-ram to main memory is purely synchronous. The assertion of **SRDY#** will latch data from **SP<8:0>** into **MUPREG**. Burst-order to and from main memory is totally transparent to burst-ram through **SA<3:2>** signals.

2.2 486 Burst-ram Operations

2.2.1 486 Burst Reads Hits

Mosel's Burst-rams support **486** burst mode operation on read hits. The first data fetch for each burst read will require two bus states. **HALE** activation will latch the address into **HADDR** register. **HALE** will be activated the same bus state as **ADS#**. The first data of any four words burst is accessed similar to any scalar read from any x 8 data rams. **BMUXC<1:0> = 00** to allow array data together with corresponding byte parity bit to bypass **MRHREG** and sent to host port **HP<8:0>**. The rising edge of **BMUXC<1>** will latch the data array entry associated with the **HADDR** register into **MRHREG**. Subsequent three remaining words of the burst read will be fetched from **MRHREG**, making very fast burst accesses possible without the use of ultra-fast (sub 10ns) srams. Burst order will be controlled by **HA<3:2>** making **486** burst order totally transparent to burst-ram. Please review figure 3 for more detail timing. For the subsequent three remaining words, **BMUXC<1:0> = 10** to allow **MRHREG** outputs to be connected to the host port **HP<8:0>**.

2.2.2 486 Scalar Write Hits

HALE will latch the address into **HADDR** register. A write hit will result in data **MWREG** to be written into the burst-ram data array. **HA<3:2>** will control which 32 bit doubleword within a quad line in the burst-ram data array is to be updated. The two write enables **HW0#,HW1#** will select which of the two banks of the burst-ram data array that is to be updated.

2.2.3 486 Read Miss Operations

Address sent out by **486** is latched into **HADDR** latch through **HALE** assertion. **HALE** is asserted the same clock as **ADS#**. A miss detection will result in **MALE** activation the following clock after **ADS#** assertion. Miss address (now in **HADDR**) will be latched into **MADDR** register. The rising edge of **BMUXC<0>** will trigger the selected replace line of the data array to be latched into **MWBREG** (see figure 4 for more detail timing). **MUPREG** will be used to hold quad fetch data from main memory. Miss data will be fetched in a **"wrapped-around"** fashion with the demand word fetched first. The demand word will then be sent to the host port through activation of **BYPASS** signal. Simultaneously, the bypass demand word is also updated into **MUPREG** register triggered by the assertion of **SRDY#**. **BMUXC<1:0>** now is set to **01** to prepare for data from **MUPREG** to be sent back to **HP<8:0>** for the demand quad burst data. The **MUPREG** now acts as a buffer register if the burst-order between the **486** host and main memory is different. **Figure 4** shows distinctively how this differential in burst-order between the two sub-systems can be compensated. Memory sub-systems using Dram nibble mode are likely to use sequenctial burst order unlike the **486**. Notice in **figure 4** that the read miss data from memory is of order **4,8,C,0** while the **486** burst order is **4,0,8,C**. As each subsequent remaining three words of the burst is read from main memory, they are bypass to the host port if the burst orders are the same. A different in burst order will result in data coming from **MUPREG**. A **QWR** can be asserted once all miss data from memory are updated into **MUPREG** register.

Each byte of the **MUPREG** is associated with a **valid** bit. As each 36 bit data word (4 data parity bits) is updated into **MUPREG** through **SRDY#** assertion, the correspondin **valid** bit will be set if **QWRWQ** is active. The inactivation of **QWRWQ** signal inhibits **valid** and **dirty** data of the same line as the miss data from being overwritten by quad fetch old data. This **valid** bit will qualify the corresponding byte of the **MUPREG** to be written into the burst-ram data array as **QWR** is asserted. After each **QWR** (quad write cycle) all **valid** and **mask** bits associated with **MUPREG** will be reset.

2.2.4 486 Write Miss Operations

486 write miss operations with **Mosel's burst-ram** is similar to **386** write miss operation in **write-back** mode. (refer to section 3.1.2.2). A write miss operation in **write-back** mode will require a minimum of three bus states (refer to fig 5). At the end of the second bus state, the rising edge of **BMUXC<0>** will trigger the latching of data from the selected replace line of data array into **MWBREG**. A **write-back** burst sequence will occur if the replace data is **valid** and **dirty**.

MUPREG will be used to hold quad fetch data from main memory. Miss data will be fetched in a"**wrapped-around**" fashion with the demand word fetched first. Each byte of the **MUPREG** is associated with a **valid** bit. As each 32 bit word is updated into **MUPREG** through **SRDY#** assertion, the corresponding **valid** bit will be set if **QWRWQ** is active. This valid bit will qualify the corresponding byte of the **MUPREG** to be written into the burst-ram data array as **QWR** is asserted. After each **QWR** (quad write cycle) all **valid** and **mask** bits associated with **MUPREG** will be reset.

Mosel's burst-ram supports "**advance**" data array update for write misses in **write-back** mode. "**Advance**" write miss processing means that write miss data can be updated into data array in burst-ram without waiting for the miss line to be fetched from main memory. Associated with each byte of the **MUPREG** is a "**mask**" bit. The corresponding "**mask**" bit will be set as an "**advance**" write occurs for the write miss data from **MWREG**. The falling edge of

HW0# or **HW1#** triggers the write data from **HP<8:0>** into **MWREG** and set the corresponding "**mask**" bit. The setting of the mask bit will inhibit the corresponding data byte in **MUPREG** from updating the data array inside the burst-ram. At reset, all four **mask** bits associated with each byte of the **MUPREG** are cleared.

2.2.5 486 Write-through (PWT cycles)

Mosel's burst-ram data-path allows **486 PWT** cycles to be supported easily. A **PWT** hit cycle is treated similar to scalar **386** writes (refer to **section 2.1.1**). A write hit will always results in data from **MWREG** to be written into the data array inside the burst-ram. **BYPASS** signal activation together with **SW#** low assertions triggers the write cycle for **MWREG** data from host port to be written to system memory bus.

A **PWT** write miss cycle is treated exactly the same as **386** write miss cycles in **write-through** mode. (refer to 2.1.2.1). A write miss will results in **MALE** activation in the next clock after **ADS#**. **MALE** activation will inhibit **MWREG** data from being written into the data array in the current clock. Data in **MWREG** will be sent to system port **SP<8:0>** by a asserting **BYPASS** and **SW#** (assert low).

A **PWT** write hit/miss cycle will require only two bus states if there is no outstanding write data already in **MWREG**.

2.2.6 486 Non-cacheable (PCD) Cycles

2.2.6.1 Non-cacheable Burst Reads

Non-cacheable burst reads are treated similar to **486** read miss (refer to section 2.2.3). The quad fetch read miss line of data are simply loaded into **MUPREG**. **MUPREG** data are then subsequently burst back to the **486**. However, in this case, no **QWR** cycle will be used. Data in **MWBREG** will not be written back to main memory. In order words, no **SW#** low assertion. Hence, the control of non-cacheable burst reads is transparent to burst-rams.

2.2.6.2 Non-cacheable Scalar read/write

Non-cacheable scalar writes are treated the same as **386** write misses in write-through mode. All non-cacheable writes will have **MALE** activation on the second bus state even if the corresponding write is a hit in the array. **MALE** inhibit write into burst-ram data array. **HW0#** or **HW1#** must be deasserted at the end of the second bus state. **BYPASS** may be asserted to sent data from **MWREG** to system port **SP<8:0>**. The low assertion of **SW#** signal allows **SP<8:0>** data to be written into system memory. Refer to section 2.1.2.1 for more detail.

Scalar reads are treated the same as **386** scalar read misses. The demand word from system port may be sent to the host port through **BYPASS** signal assertion with **SW#** deasserted.

Hence, all non-cacheable cycles are totally supported by **Mosel's burst-ram** data-path architecture.

3.0 Multiple replacement cycles

Mosel's Burst-ram supports cache systems with block size larger than four 32 bit double words. Attached to the least significant two bits of **MADDR** is a two bit wrapped around counter. Figure 9. shows the timing of a read miss with multiple replacement cycles. The demand line of read miss data is fetched first from system memory. Data of the replace line is then latched into **MWBREG** to be written to system memory. **MPINC** signal is then activated to direct **MADDR** to access the neighboring data line to be replaced. The manner in which the least significant two bits of **MADDR** changes as a function of **MPINC** activation depends on the initial address. The change order is similar to **486** except in this case, **MADDR<5:4>** signifies the replace data line within a block rather than a 32 bit doubleword within a line.

The table below illustrates the change order of **MADDR<5:4>** as a function of **MPINC**.

	MADDR<5:4>				MPINC
initial address	00	01	10	11	starting address
2nd replacement	01	00	11	10	after 1st assertion
3rd replacement	10	11	00	01	after 2nd assertion
4th replacement	11	10	01	00	after 3rd assertion

Cache systems with block size of eight 32 bit doublewords need only to activates **MPINC** once for the second data replacement cycle. Cache system with block size of sixteen 32 bit doublewords will activate **MPINC** three times.

Each data replacement cycle begins with **MPINC** assertion. **BMUXC<0>** is then set to zero. An active assertion of **BMUXC<0>** allows replace data read from the burst-ram data array to be latched into **MWBREG**. **SW#** assertion allows the data in **MWBREG** to be written back to system memory through **SP<8:0>**. The burst order of replace data will be controlled by **SA<3:2>**.

With **MWBREG** and **MUPREG**, **Mosel's** burst-ram allows the demand read miss data to be fetch first from system memory. The processor does not have to wait for the completion of the replacement cycles. This arrangement makes replacement cycles to be totally hidden from the processor. For cache system with large line size (say 16 double-words), **Mosel's Burst-rams** eliminates this large replacement penalty.

3.0 BURST-RAM PIN-OUT DEFINITIONS

ADDR<14:4>	inputs	386/486 local address bus. These bits address one of 2048 entries of 32 bit double-word in each burst-ram data array.
HALE	input	Burst-ram input address latch enable. This signal activates the same time as ADS#. HALE opens HADDR level latch to ADDR.
MALE	input	Burst-ram miss address latch enable. This signal activates one CLK after HALE activation if either a read/write miss has been detected. MALE signal opens MADDR level latch to HADDR latch output. Simultaneous assertion of MALE together with HW0# or HW1# will inhibit writing to data array inside Burst-ram.
RESET	input	Reset should be active for at least four CLK clocks for burst-ram to complete reset. Reset will clear all mask and valid bits.
BMUXC<1:0>	inputs	These two bits control the burst-ram BMUX. The rising edge of BMUXC<0> results in burst-ram data entry associated with

MADDR<14:4> to be latched into MWBREG<35:0>. These data will be written to main memory if they are **dirty** and **valid**. The rising edge of BMUXC<1> results in burst-ram data entry associated with HADDR<14:4> to be latched into MRHREG<35:0>.
For multiple replacement cycles, BMUXC<0> must be asserted on the second clock after MPINC is asserted to allow burst-ram data entry associated with MADDR<14:4> to be latched into MWBREG<35:0>.
The control bits definition are as follows:-

BMUXC<1:0>	SOURCE
0,0	M[FADDR<14:4>]<35:0>
0,1	MUPREG<35:0>
1,0	MRHREG<35:0>
1,1	illegal

HP<8:0>	inputs/ outputs	Byte-wide data input/output to and from host processor (386/486) with data parity. HP<8> is used if cache system support data parity. HP<8> can be disconnected if data parity is not used.

QWRWQ	**input**	Quad write word qualify Active assertion of this signal qualify the miss fetch word associated with **Srdy#** to be written into the data array of the **Burst-ram.**
HA<3:2>	**inputs**	These two bits indicates the word address within a quad word. These bits are part of the address bit associated with the data at the host port.
HW0#, HW1#	**inputs**	**HW0#** is host port write enable for bank0 and **HW1#** is host port write enable for bank1. A low assertion of either **HW0#** or **HW1#** (**HW#**) indicates the corresponding bus cycle generated by the host is a write cycle. Else a read operation is assumed. The falling edge of either of these two signals triggers the latching of **HP<8:0>** data into **MWREG. HW0#** or **HW1#** will be asserted during the second bus state of any write cycles. These two signals cannot be both active simultaneously.
G0#, G1#	**inputs**	Host port output enables **G0#** will enable data from bank0 for the corresponding cycle, while **G1#** will enable data from bank1. Both signals cannot be active simultaneously. **G0#** or

		G1# will be asserted only during the second bus state of a bus cycle.
SP<8:0>	**inputs/ outputs**	Byte-wide data input/output to and from main memory with data parity.
SA<3:2>	**inputs**	These two bits indicates the word address within a quad word. These address bits are part of the address associated with the data at the system port.
SRDY#	**input**	A low activation of this signal indicates that the data read from main memory is ready. The rising edge of this signal triggers the system port data **SP<8:0>** to be latched into **MUPREG** in the burst-ram at the word location selected by **SA<3:2>**.
SW#	**input**	System port output enable. A low assertion of **SW#** indicates that the system port will be performing a write operation to main memory. For burst-write, **SW#** will be asserted for all four write data transactions. A high assertion of this signal assumes a read operation at the system port.
QWR	**input**	Quad write Activation of quad write

		results in 36 bit data residing in **MUPREG<35:0>** to be written into burst-ram data array entry pointed to by the address in **MADDR** and the burst-ram internal bank select logic. **QWR** overrides **BMUXC** control. Activation of **QWR** will reset all valid bits associated with each data byte in **MUPREG**.
BYPASS	input	This signal connect host post data **HP<8:0>** to system port **SP<8:0>**.
SELECT#	input	Active low assertion signal. This signal indicates that the corresponding burst-ram is selected. An high assertion of this signal will results in all data output pins to be tri-state. **Select#** pin allows cache systems expansion with more **burst-rams**.
MPINC	input	Multiple replacement cycle **MADDR** address increment control. Assertion of this signal allows **MADDR** to be change to the next neighboring replacement line to be accessed from the burst-ram data array.
CLK	input	This **CLK** signal is the same as the **CLK** line in **486**

		systems. For systems using **386**, this signal will be connected to **CLK2** as input. An internal divided by two circuit will set the burst-ram internal clock to match that of the **386**.
486/386#	**input**	This signal indicates to the **burst-ram** that it is in a **386** system when this signal is strapped active else a **486** system environment is assumed.

4.0 BURST-RAM AHDL FLOW

This section shows a tentative **AHDL** flow of **Mosel Burst-ram**. The **AHDL** assume only a single bank. A more accurate flow will be published once the behavior code of the Burst-ram has been proven through simulation. Nonetheless, the flow below should give a concise description of the architecture.

Nomenclature is listed as follows:-

symbols	meanings
!	vector concatenation
^	logical "**and**"
~	logical "**negate**"
+	logical "**or**"
xor	logical exclusive "or"
𝔉I X	Falling edge detection of signal "X"
ℜI X	Rising edge detection of signal "X"
{}	comments
#	active low assertion
y * Z	y signifies data path Z signifies corresponding control
<---	latch operand at rising edge of clock unless specified by ℜIX or 𝔉IX
<--- (ℜI X)	latch operand at rising edge of signal X
<--- (𝔉I X)	latch operand at falling edge of signal X

4.1 AHDL showing Burst-ram datapath operations

AHDL below currently assume only one data bank.

Multiple replacement logic is included in this flow.

BURST-RAM ARRAY ORGRANIZATION: 2K * 9

INPUTS:	BYPASS	{CONNECT HP<8:0> TO SP<8:0>}
	QWRWQ	{ QUAD WRITE WORD QUALIFY}
	HP<8:0>	{ HOST READ/WRITE PORT}
	HA<3:2>	{ HOST PORT DOUBLE-WORD ADDRESS WITHIN A LINE}
	HW#	{ HOST PORT WRITE ENABLE}
	MALE	{ MISS ADDRESS LATCH ENABLE}
	BMUXC<1:0>	{ BMUX 4:1 MUX CONTROL}
	SW#	{ SYSTEM PORT WRITE TO MAIN MEMORY}
	SRDY#	{ SYSTEM PORT DATA READY}
	SA<3:2>	{ SYSTEM PORT DOUBLE-WORD ADDRESS WITH A LINE}
	QWR	{ QUAD WRITE TO ARRAY}
	SP<8:0>	{ SYSTEM READ/WRITE PORT}
	G#	{ OUTPUT ENABLE FOR HOST PORT}
	SELECT#	{ CHIP SELECT FOR BURST-RAM}
	MPINC	{MULTIPLE REPLACEMENT CYCLE ACTIVATE}
	CCLK	{EXTERNAL CLOCK INPUT}

OUTPUTS:	HP<8:0>	{ HOST READ/WRITE PORT}
	SP<8:0>	{ SYSTEM READ/WRITE PORT}
REGISTERS:	M[2048,36]	{DATA ARRAY 2Kx36, assume single bank}
	MUPREG<35:0>	{MEMORY UPDATE REGISTER}
	MWBREG<35:0>	{MEMORY WRITE-BACK REGISTER}
	MWREG<8:0>	{HOST BUFFER-WRITE REGISTER}
	MRHREG<35:0>	{MEMORY READ HOLD DATA REGISTER}
	VALID3	{VALID BIT FOR MUPREG<35:27>}
	VALID2	{VALID BIT FOR MUPREG<26:18>}
	VALID1	{VALID BIT FOR MUPREG<17: 9>}
	VALID0	{VALID BIT FOR MUPREG< 8: 0>}
	MASK3	{MASK BIT FOR MUPREG<35:27>}
	MASK2	{MASK BIT FOR MUPREG<26:18>}
	MASK1	{MASK BIT FOR MUPREG<17: 9>}
	MASK0	{MASK BIT FOR MUPREG< 8: 0>}
	SPSELPM	{SYSTEM PORT SELECT DUE TO PENDING MISS PROCESSING}
	BMUXC0D	{DELAY BMUXC<0> INPUT FOR POSITIVE EDGE DETECTION}
	BMUXC1D	{DELAY BMUXC<1> INPUT FOR POSITIVE EDGE DETECTION}

HP<8:0>= BMUX<35:27> *

```
                    (HA<3:2> = 1,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
                    BMUX<26:18>      *
                    (HA<3:2> = 1,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
                    BMUX<17: 9 >     *
                    (HA<3:2> = 0,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
                    BMUX< 8: 0>      *
                    (HA<3:2> = 0,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
                    SP<8:0>          *
                    (BYPASS ^ SW# ^ ~SELECT#)   ;

BMUX<35:0>=     (MRHREG<35:0>) * (BMUXC<1:0> = 1,0)
                (MUPREG(35:0>) * (BMUXC<1:0> = 0,1)
                (M[FADDR<14:4>]<35:0>) * (BMUXC<1:0> = 0,0);


//* BMUXC<1> , BMUXC<0>   positive edge detection logic

BMUXC1D         <------     BMUXC<1>;

BMUXC0D         <------     BMUXC<0>;

MRHREG<35:0>    <------     M[FADDR<14:4>]<35:0> *
                            (BMUXC<1> ^ ~BMUXC1D)      ;
MWBREG<35:0>    <------     M[FADDR<14:4>]<35:0> *
                            (BMUXC<0> ^ ~BMUXC0D)      ;

SP<8:0> =   MWBREG<35:27>  *
            (SA<3:2> = 1,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG<26:18>  *
            (SA<3:2> = 1,0 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG<17: 9>  *
            (SA<3:2> = 0,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG< 8: 0>  *
            (SA<3:2> = 0.0 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWREG<8:0>     *
            (BYPASS ^ ~SW# ^ ~SPSEL)                ;
```

```
MUPREG<35:27>  <----  SP<8:0>      *
                      (SA<3:2> = 1,1 ^ SRDY ^ SW#)  ;
MUPREG<26:18>  <----  SP<8:0>      *
                      (SA<3:2> = 1,0 ^ SRDY ^ SW#)  ;
MUPREG<17: 9 > <----  SP<8:0>      *
                      (SA<3:2> = 0,1 ^ SRDY ^ SW#)  ;
MUPREG< 8:0 >  <----  SP<8:0>      *
                      (SA<3:2> = 0,0 ^ SRDY ^ SW#) ;

MWREG<  8:0  >  <----  HP<8:0>     *      ( HW# ^ ~CCLK) ;


VALID3     <---   (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 1,1 ^  QWRWQ ^ SRDY);
VALID2     <---   (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 1,0 ^ QWRWQ ^ SRDY);
VALID1     <---   (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 0,1 ^ QWRWQ ^ SRDY);
VALID0     <---   (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 0,0 ^  QWRWQ ^ SRDY);

MASK3      <---   (~HW# + MALE) *
                  (HA<3:2> = 1,1  ^ ~MALE ^ ~HW#);
MASK2      <---   (~HW# + MALE) *
                  (HA<3:2> = 1,0 ^ ~MALE ^ ~HW#);
MASK1      <---   (~HW# + MALE) *
                  (HA<3:2> = 0,1  ^ ~MALE ^ ~HW#);
MASK0      <---   (~HW# + MALE) *
                  (HA<3:2> = 0,0  ^ ~MALE ^ ~HW#);
```

```
M[FADDR<14:4>]<35:27> <---  MWREG<8:0>      *
                            (HA<3:2> = 1,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<35:27> <---  MUPREG<35:27>   *
                            (VALID3 ^ QWR ^ ~MASK3);

M[FADDR<14:4>]<26:18> <---  MWREG<8:0>      *
                            (HA<3:2> = 1,0 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<26:18> <---  MUPREG<26:18>   *
                            (VALID2 ^ QWR ^ ~MASK2);

M[FADDR<14:4>]<17: 9> <---  MWREG<8:0>      *
                            (HA<3:2> = 0,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<17: 9> <---  MUPREG<17: 9>   *
                            (VALID1 ^  QWR ^ ~MASK1);

M[FADDR<14:4>]< 8: 0> <---  MWREG<8: 0>     *
                            (HA<3:2> = 0,0 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]< 8: 0> <---  MUPREG<8: 0>    *
                            (VALID0 ^ QWR ^ ~MASK0);

SPSELPM   <---   (1) *
                 ((~SELECT# ^ MALE) + (~MALE ^ MPINC)) ;
SPSELPM   <---   (0)    *
                 (BYPASS ^ ~SW# ^ ~SELECT#)              ;
SPSEL      =     (SPSELPM) +
                 ((~SELECT#) ^ (BYPASS ^ ~SW#))          ;
```

4.2 AHDL showing burst-ram address path operations

```
INPUTS:        ADDR<14:4>      { HOST PROCESSOR ADDRESS}
               HALE            { ADDRESS LATCH ENABLE }
               MALE            { MISS ADDRESS LATCH
                                 ENABLE }
               QWR             { QUAD WRITE }
               MPINC           { MULTIPLE REPLACEMENT
                                 CYCLE ACTIVATE}
               CCLK            {EXTERNAL CLOCK INPUT}

REGISTERS:     HADDR<14:4>     { HIT ADDRESS REGISTER}
               MADDR<14:4>     { MISS ADDRESS REGISTER}


FADDR<14:4> =  ADDR<14:4>      *     HALE
               HADDR<14:4>     *     (~HALE ^ ~QWR)
               MADDR<14:4>     *     (~HALE ^  QWR)

HADDR<14:4>    <---   ADDR<14:4> * (HALE ^ ~CLK);
//*     LEVEL LATCH

MADDCNTLE =    MALE  +  MPINC                        ;
MADDCNTIN =    ~BMADDCNT ^    ~MALE                  ;
MADDCNT        <---   MADDCNTIN * MADDCNTLE          ;
//* MULTIPLE  REPLACEMENT CYCLE CONTROL

MADDR<14:6>    <---   HADDR<14:6> * MALE             ;

MADDRIN<5> =   (MALE ^ ADDR<5>)        +
               (~MALE ^ MADDR<5> xor MADDCNT)        ;
MADDRIN<4> =   (MALE ^ ADDR<4>)        +
               (~MALE ^ MADDR<4>)                    ;
MADDR<5:4>     <---   BMADDRIN<5:4> * MADDCNTLE      ;
```

4.3 BURST-RAM OPERATION MODEL

4.3.1 OPERATION : READ HIT - BURST READ HIT

ACTIONS:

```
MRHREG<35:0>     <---    M[FADDR<14:4>]<35:0>  *
                         (BMUXC<1> ^ ~BMUX1D) ;

HP<7:0>=     BMUX<35:27> *
             (HA<3:2> = 1,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
             BMUX<26:18> *
             (HA<3:2> = 1,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
             BMUX<17: 9> *
             (HA<3:2> = 0,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
             BMUX< 8: 0> *
             (HA<3:2> = 0,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#);

BMUX<35:0> = MRHREG<35:0> * (BMUXC<1:0> = 0,0) ;
                                   { 1ST DATA OPERAND READ}

BMUX<35:0> = MRHREG<35:0> * (BMUXC<1:0> = 1,0) ;
                                   { REMAINING 3 DATA READS}
```

4.3.2 OPERATION: READ HIT - SCALAR READ HIT

ACTIONS:

```
HP<8:0> =    BMUX<35:27> *
             (HA<3:2> = 1,1 ^ ~BYPASS  ^ ~SELECT# ^ ~G#)
             BMUX<26:18> *
             (HA<3:2> = 1,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
             BMUX<17: 9> *
             (HA<3:2> = 0,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
             BMUX< 8: 0> *
             (HA<3:2> = 0,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#) ;


BMUX<35:0> =     M[FADDR<14:4>]<35:0> * (BMUXC<1:0> = 0,0) ;
```

4.3.3 OPERATION: WRITE HIT - SCALAR WRITE HIT

ACTIONS (WRITE-BACK MODE):

```
MWREG<8:0>              <---- HP<8:0>      *
                              ( HW# ^ ~CCLK)   ;

M[FADDR<14:4>]<35:27><---- MWREG<8:0>      *
                           (HA<3:2> = 1,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<26:18><---- MWREG<8:0>      *
                           (HA<3:2> = 1,0 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<17: 9><---- MWREG<8:0>      *
                           (HA<3:2> = 0,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]< 8: 0><---- MWREG<8:0>      *
                           (HA<3:2> = 0,0 ^ ~HW# ^ ~QWR);
```

ACTIONS (WRITE-THROUGH MODE):

```
MWREG<8:0>              <--- HP<8:0>      *
                             ( HW# ^ ~CCLK)            ;

M[FADDR<14:4>]<35:27><---  MWREG<8:0>      *
                           (HA<3:2> = 1,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<26:18><---  MWREG<8:0>      *
                           (HA<3:2> = 1,0 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]<17: 9><---  MWREG<8:0>      *
                           (HA<3:2> = 0,1 ^ ~HW# ^ ~QWR);
M[FADDR<14:4>]< 8: 0><---  MWREG<8:0>      *
                           (HA<3:2> = 0,0 ^ ~HW# ^ ~QWR);

SP<8:0>      =      MWREG<8:0>      *
                    (BYPASS ^ ~SW# ^ ~SPSEL)          ;
```

4.3.4 OPERATIONS: READ MISS

ACTIONS:

```
{REPLACE DATA}
MWBREG<35:0>     <----  M[FADDR<14:4>]<35:0>  *
                        (BMUXC<0> ^ ~BMUXC0D)      ;

{MISS WORD READ}
MUPREG<35:27>   <----  SP<8:0>       *
                       (SA<3:2> = 1,1 ^ (SRDY#) ^ SW#);
MUPREG<26:18>   <----  SP<8:0>       *
                       (SA<3:2> = 1,0 ^ (SRDY#) ^ SW#);
MUPREG<17: 9>   <----  SP<8:0>       *
                       (SA<3:2> = 0,1 ^ (SRDY#) ^ SW#);
MUPREG< 8: 0>   <----  SP<8:0>       *
                       (SA<3:2> = 0,0 ^ (SRDY#) ^ SW#);

{VALID BIT UPDATE}
VALID3    <----  (~SRDY# ^ ~MALE)       *
                 (SA<3:2> = 1,1 ^ QWRWQ ^ SRDY);
VALID2    <----  (~SRDY# ^ ~MALE)       *
                 (SA<3:2> = 1,0 ^ QWRWQ ^ SRDY);
VALID1    <----  (~SRDY# ^ ~MALE)       *
                 (SA<3:2> = 0,1 ^ QWRWQ ^ SRDY);
VALID0    <----  (~SRDY# ^ ~MALE)       *
                 (SA<3:2> = 0,0 ^  QWRWQ ^ SRDY);


{UPDATE  CACHE MEMORY ARRAY}
M[FADDR<14:4>]<35:27>      <----  MUPREG<35:27>   *
                                  (VALID3 ^ QWR ^ ~MASK3);
M[FADDR<14:4>]<26:18>      <----  MUPREG<26:18>   *
                                  (VALID2 ^ QWR ^ ~MASK2);
M[FADDR<14:4>]<17: 9>      <----  MUPREG<17: 9>   *
                                  (VALID1 ^ QWR ^ ~MASK1);
M[FADDR<14:4>]< 8: 0>      <----  MUPREG< 8: 0>   *
                                  (VALID0 ^ QWR ^ ~MASK0);
```

```
{SYSTEM PORT SELECT LOGIC}
SPSELPM  <----    (1)  *
                  ((~MALE ^ MPINC) + (~SELECT# ^ MALE));
SPSELPM  <----    (0)  *
                  (BYPASS ^ ~SW# ^ ~SELECT#)              ;
SPSEL        =    SPSELPM +
                  (~SELECT# ^ BYPASS ^ ~SW#)              ;


{REPLACE DATA WRITE BACK TO MAIN MEMORY}
SP<8:0> =     MWBREG<35:27>  *
              (SA<3:2> = 1,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
              MWBREG<26:18>  *
              (SA<3:2> = 1,0 ^ ~SW# ^ SPSEL ^ ~BYPASS)
              MWBREG<17:  9>  *
              (SA<3:2> = 0,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
              MWBREG< 8:  0>  *
              (SA<3:2> = 0,0 ^ ~SW# ^ SPSEL ^ ~BYPASS)         ;

{BURST READ BACK TO HOST}

BMUX<35:0> = MUPREG<35:0> * (BMUXC<1:0> = 0,1)    ;

HP<8:0> =     SP<8:0> *
              (BYPASS ^ SW# ^ ~SELECT#) ;
              { IF SAME BURST ORDER}

HP<8:0> =     BMUX<35:27>       *
              (HA<3:2> = 1,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
              BMUX<26:18>       *
              (HA<3:2> = 1,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
              BMUX<17:  9>      *
              (HA<3:2> = 0,1 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
              BMUX<  8:  0>     *
              (HA<3:2> = 0,0 ^ ~BYPASS ^ ~SELECT# ^ ~G#)
                          ; { IF DIFFERENT BURST-ORDER}
```

4.3.5 OPERATIONS: WRITE MISS

ACTIONS

```
{REPLACE DATA}
MWBREG<35:0>    <----  M[FADDR<14:4>]<35:0>  *
                       (BMUXC<0> ^ ~BMUXC0D);

{MISS WORD READ}
MUPREG<35:27>   <----  SP<8:0>       *
                       (SA<3:2> = 1,1 ^  (SRDY#) ^ SW#);
MUPREG<26:18>   <----  SP<8:0>       *
                       (SA<3:2> = 1,0 ^  (SRDY#) ^ SW#);
MUPREG<17: 9>   <----  SP<8:0>       *
                       (SA<3:2> = 0,1 ^  (SRDY#) ^ SW#);
MUPREG< 8: 0>   <----  SP<8:0>       *
                       (SA<3:2> = 0,0 ^  (SRDY#) ^ SW#);

{VALID BIT UPDATES}
VALID3     <----  (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 1,1 ^ QWRQW ^ SRDY);
VALID2     <----  (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 1,0  ^ QWRQW ^ SRDY);
VALID1     <----  (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 0,1  ^ QWRQW ^ SRDY);
VALID0     <----  (~SRDY# ^ ~MALE) *
                  (SA<3:2> = 0,0  ^ QWRQW ^ SRDY);

{UPDATE  TO CACHE MEMORY ARRAY}
M[FADDR<14:4>]<35:27><----  MUPREG<35:27> *
                            (VALID3 ^ QWR ^ ~MASK3)   ;
M[FADDR<14:4>]<26:18><----  MUPREG<26:18> *
                            (VALID2 ^ QWR ^ ~MASK2)   ;
M[FADDR<14:4>]<17: 9><----  MUPREG<17: 9> *
                            (VALID1 ^ QWR ^ ~MASK1)   ;
M[FADDR<14:4>]< 8: 0><----  MUPREG< 8: 0> *
                            (VALID0 ^ QWR ^ ~MASK0)   ;
```

```
{ SYSTEM PORT SELECT LOGIC}
SPSELPM <---     (1) *
                 ((~SELECT# ^  MALE) + ( ~MALE ^ MPINC));

SPSELPM <---     (0) *
                 (BYPASS ^ ~SW# ^ ~SELECT#)                 ;

SPSEL       =    SPSELPM +
                 (~SELECT# ^ BYPASS ^ ~SW#)                 ;


{REPLACE DATA BACK TO MAIN FOR WRITE BACK}
SP<8:0> =   MWBREG<35:27> *
            (SA<3:2> = 1,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG<26:18>  *
            (SA<3:2> = 1,0 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG<17:  9>  *
            (SA<3:2> = 0,1 ^ ~SW# ^ SPSEL ^ ~BYPASS)
            MWBREG< 8:  0>  *
            (SA<3:2> = 0,0 ^ ~SW# ^ SPSEL ^ ~BYPASS)        ;

{LATCH WRITE DATA FROM HOST PROCESSOR}
MWREG<8:0>      <---   HP<8:0>     *     ( HW# ^ ~CCLK)    ;

{MASK BIT OPERATIONS}
MASK3       <---- (~HW# + MALE)   *
                  (HA<3:2> = 1,1  ^ ~MALE ^ ~HW#) ;
MASK2       <---- (~HW# + MALE)   *
                  (HA<3:2> = 1,0  ^ ~MALE ^ ~HW#) ;
MASK1       <---- (~HW# + MALE)   *
                  (HA<3:2> = 0,1  ^ ~MALE ^ ~HW#) ;
MASK0       <---- (~HW# + MALE)   *
                  (HA<3:2> = 0,0  ^ ~MALE ^ ~HW#) ;

{WRITE  HOST DATA TO CACHE MEMORY}
M[FADDR<14:4>]<35:27><----  MWREG<8:0>       *
                            (HA<3:2> = 1,1 ^ ~HW# ^ ~ QWR) ;
M[FADDR<14:4>]<26:18><----  MWREG<8:0>       *
                            (HA<3:2> = 1,0 ^ ~HW# ^ ~QWR) ;
M[FADDR<14:4>]<17:  9><---- MWREG<8:0>       *
```

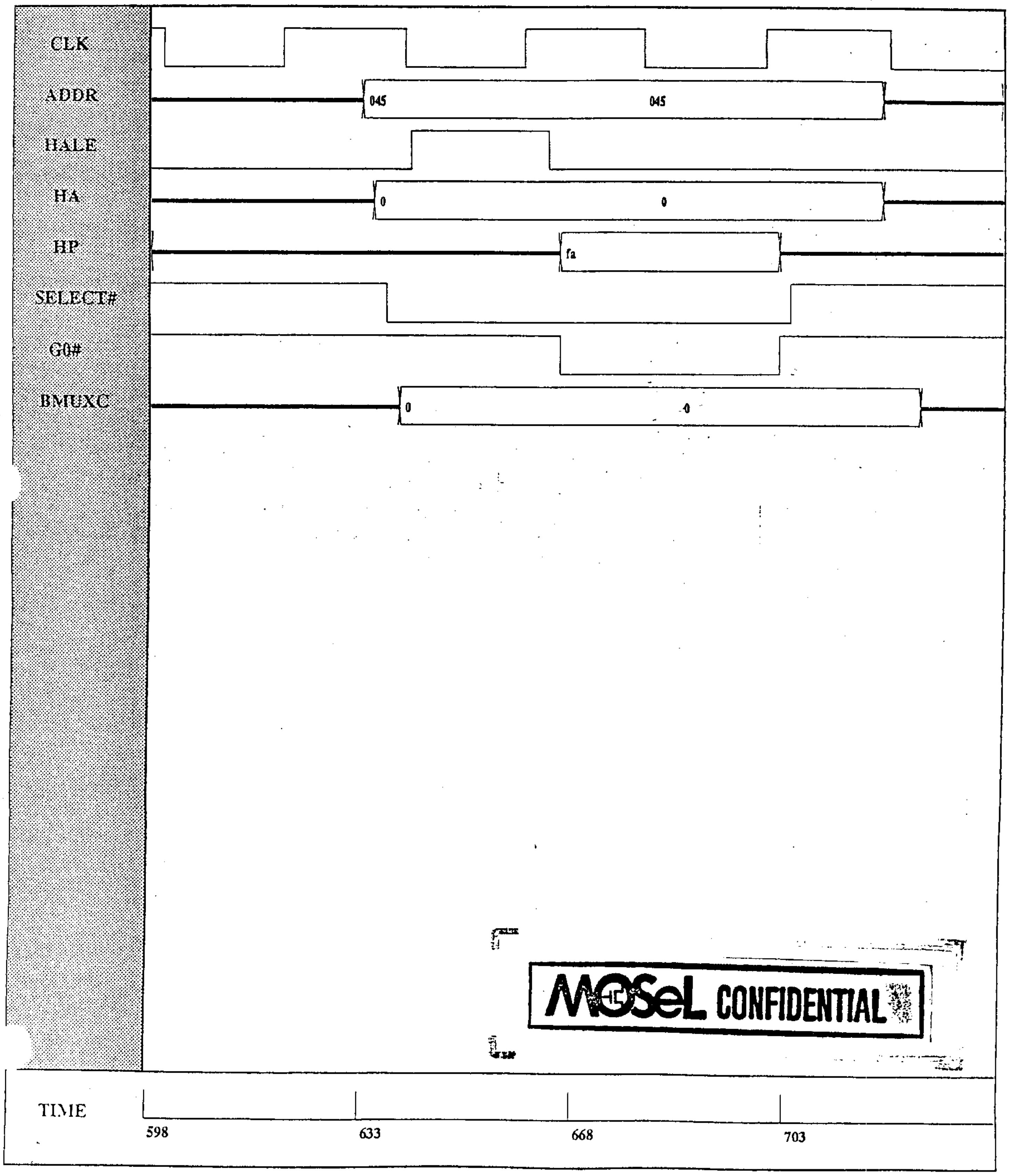
```
                                    (HA<3:2> = 0,1 ^ ~HW# ^ ~QWR) ;
M[FADDR<14:4>]< 8: 0><----  MWREG<8:0>         *
                                    (HA<3:2> = 0,0 ^ ~HW# ^ ~QWR) ;
```

Gateway

Header: Fig.1 Host port scalar read

User: Jason Lin

Date: Feb 13, 1990 14:29:50 Time Scale From: 598 To: 739 Page: 1 of 1

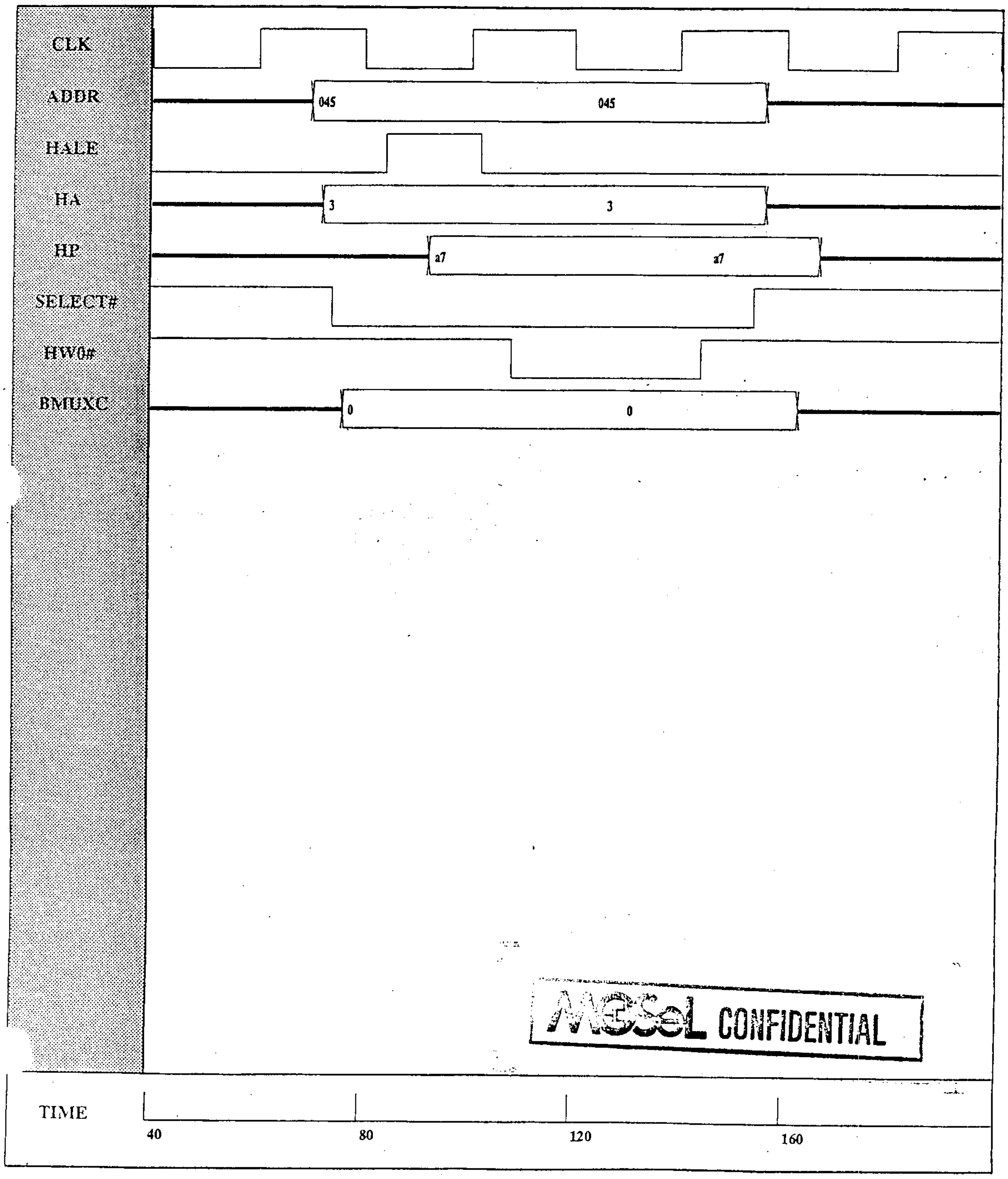


Gateway

Header: Fig. 2 Host port scalar write

User: Jason Lin

Date: Feb 13, 1990 13:49:44 Time Scale From: 40 To: 200 Page: 1 of 1

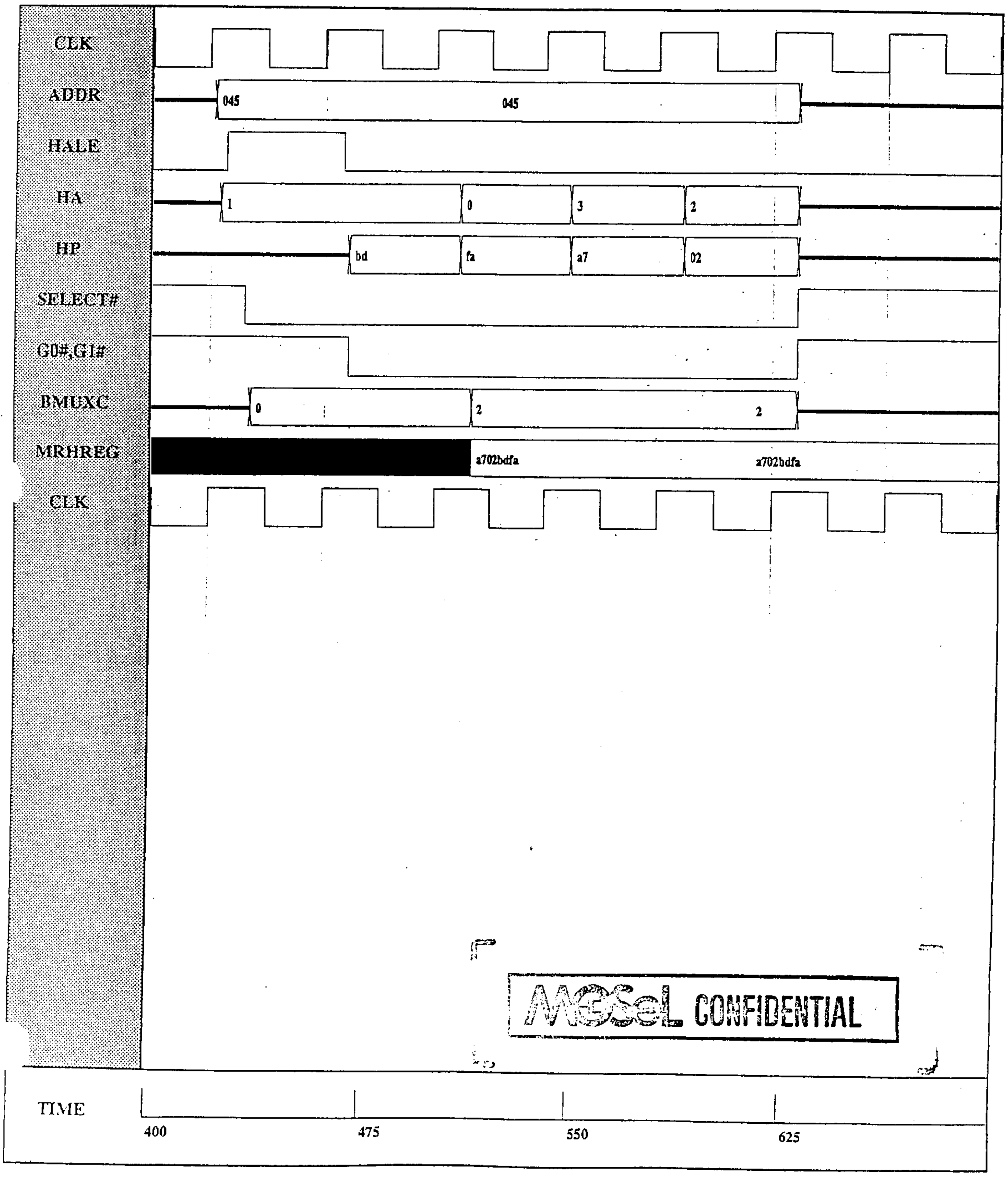


Gateway

Header: Fig.3 Host port burst read

User: Jason Lin

Date: Feb 16, 1990 12:10:18 Time Scale From: 400 To: 700 Page: 1 of 1



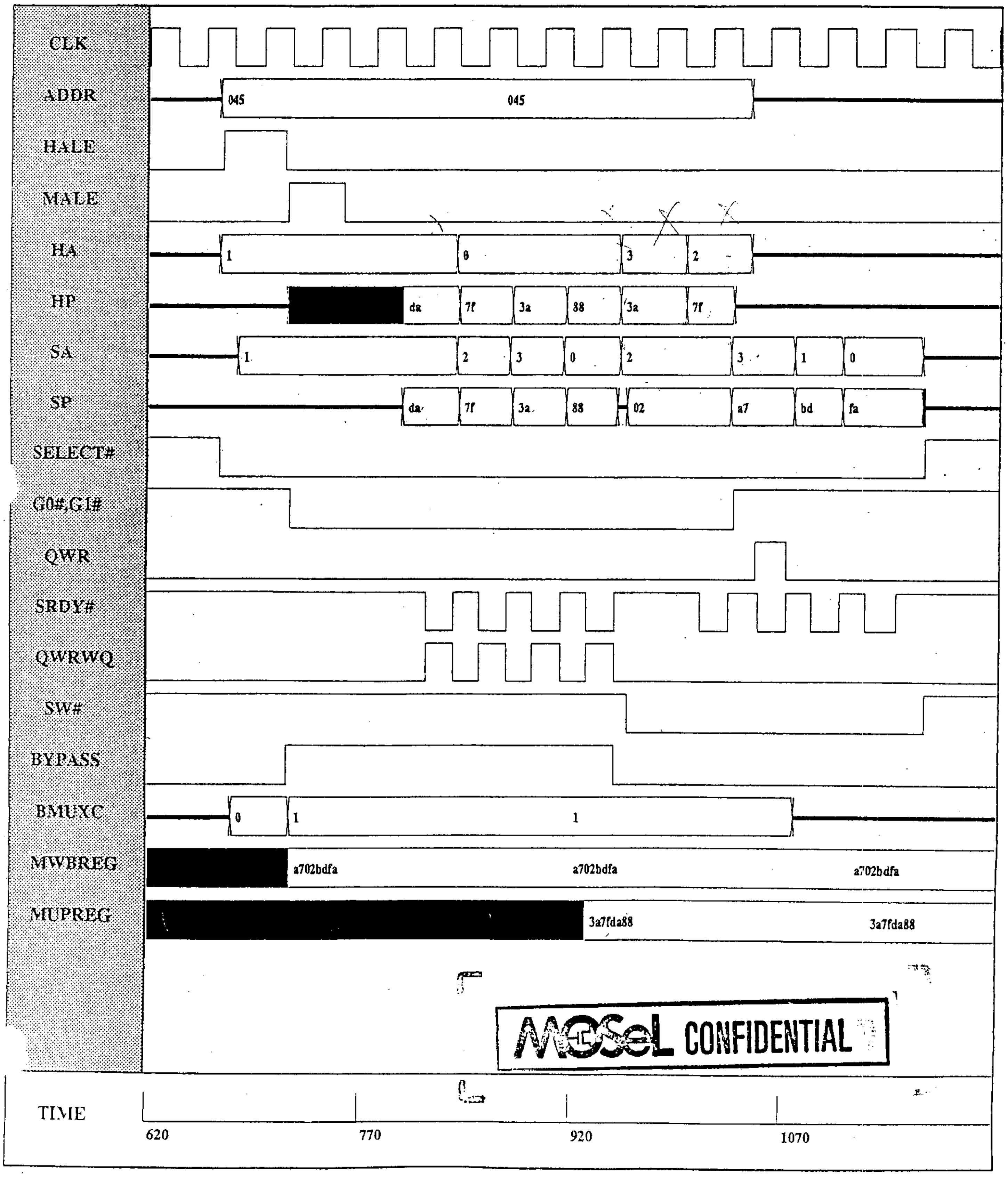
MOSEL CONFIDENTIAL

Gateway

Header: Fig.4 Read miss processing (burst reordering)

User: Jason Lin

Date: Feb 16, 1990 09:10:59 Time Scale From: 620 To: 1220 Page: 1 of 1



Gateway

Header: Fig.5 Write-back mode write miss processing

User: Jason Lin

Date: Feb 16, 1990 10:32:27 Time Scale From: 620 To: 1260 Page: 1 of 1

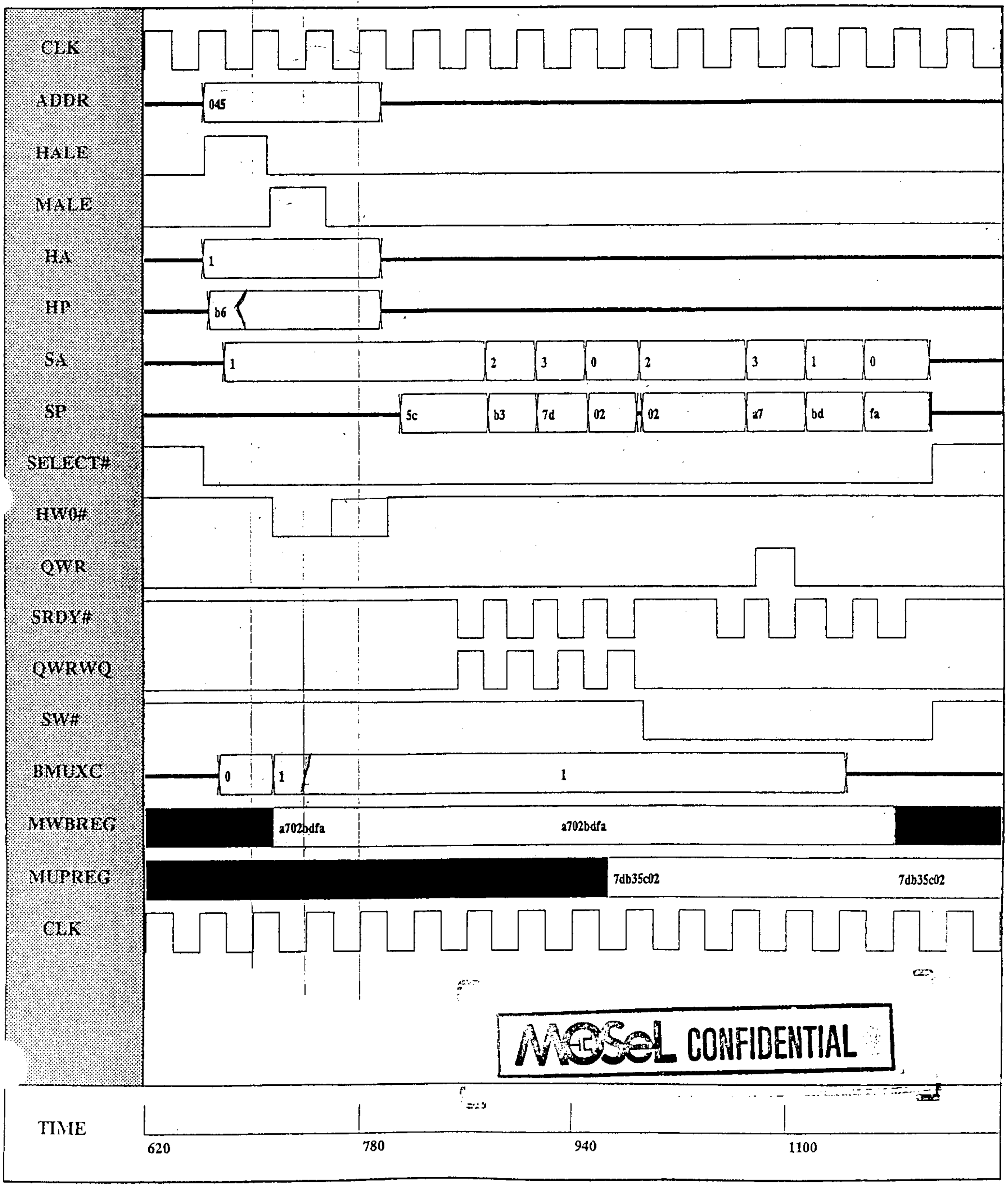
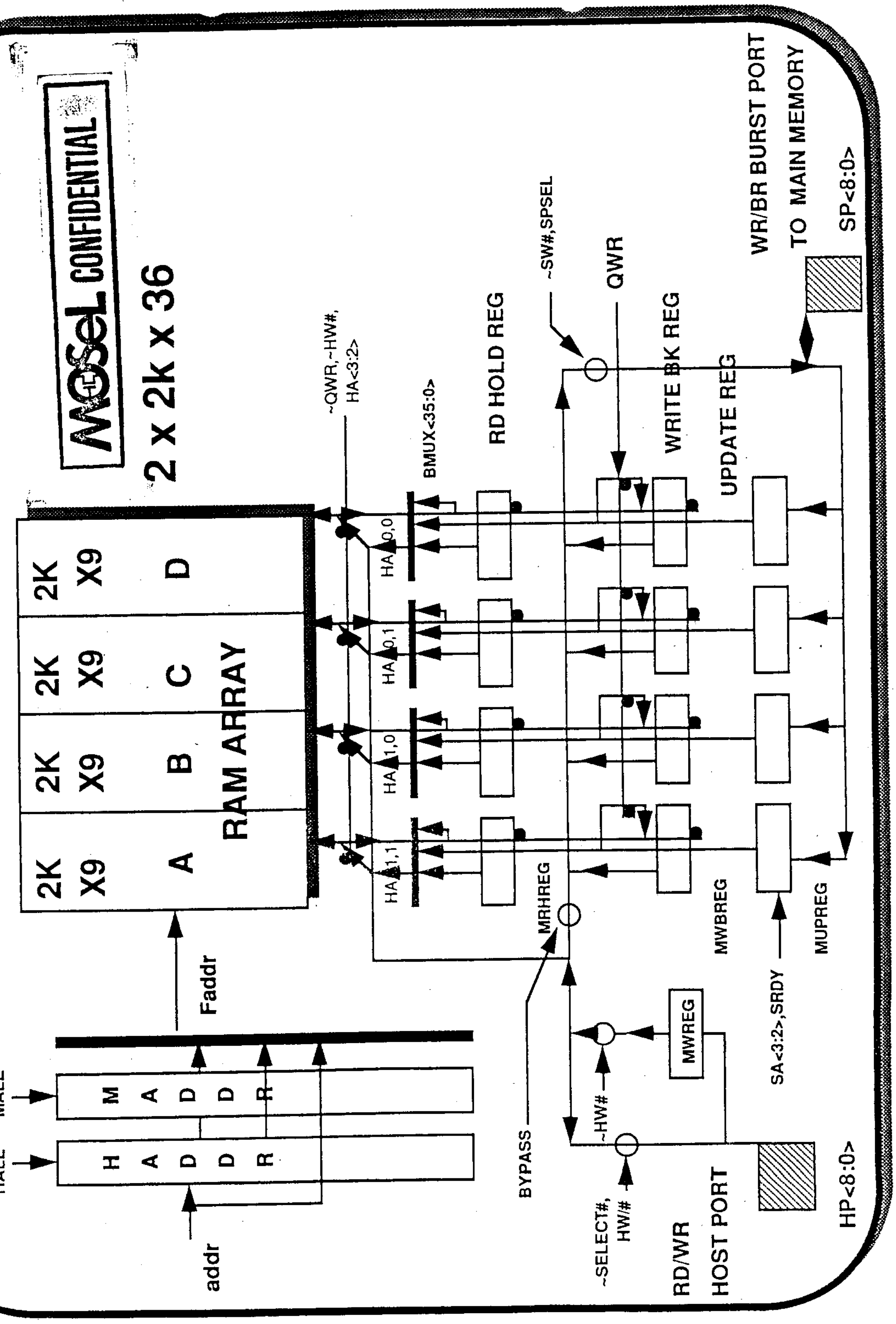


FIG 6: BURST-RAM rev 1.3

2 x 2k x 36

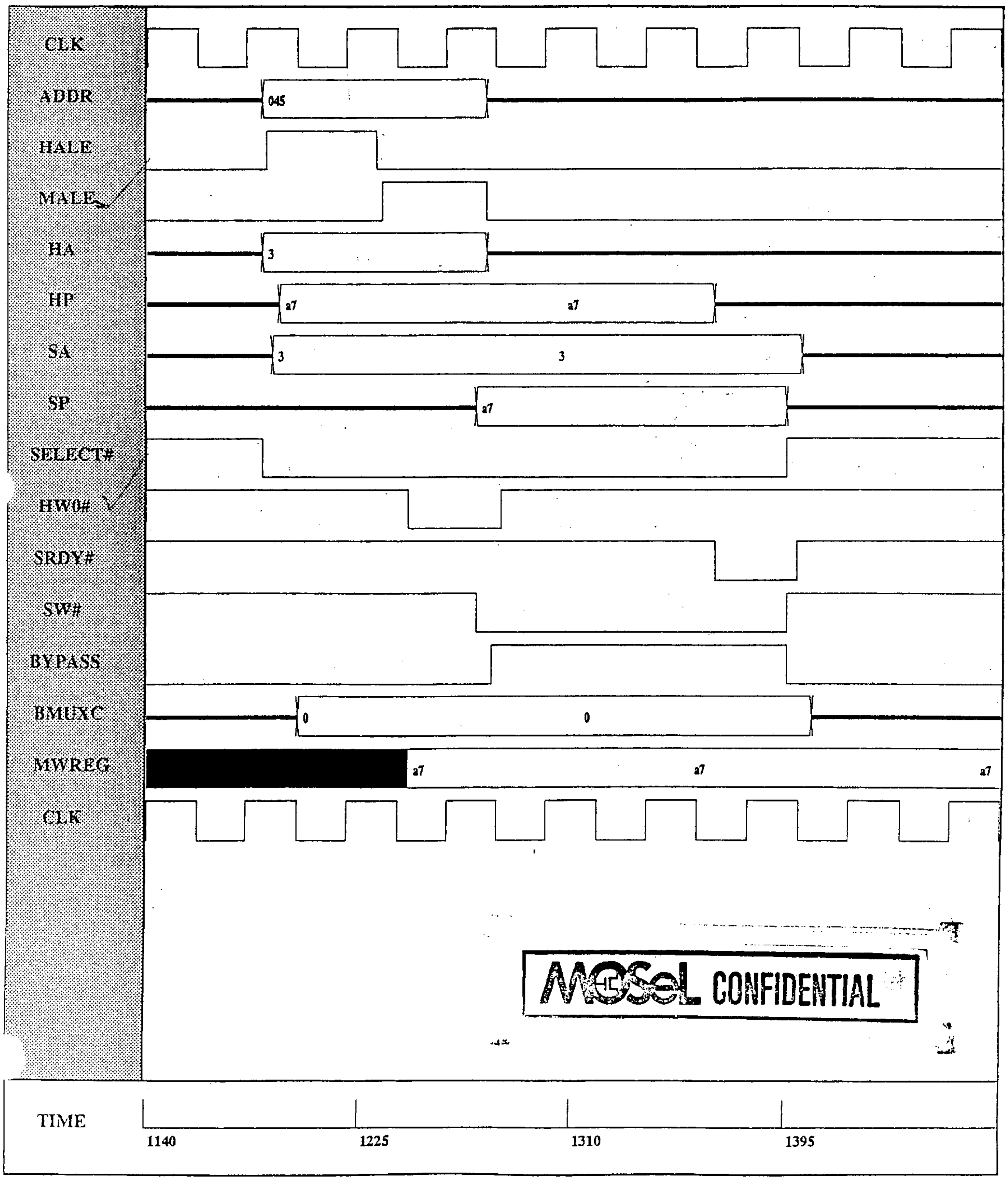


Gateway

Header: Fig.7 Write through mode write miss

User: Jason Lin

Date: Feb 16, 1990 11:25:45 Time Scale From: 1140 To: 1480 Page: 1 of 1

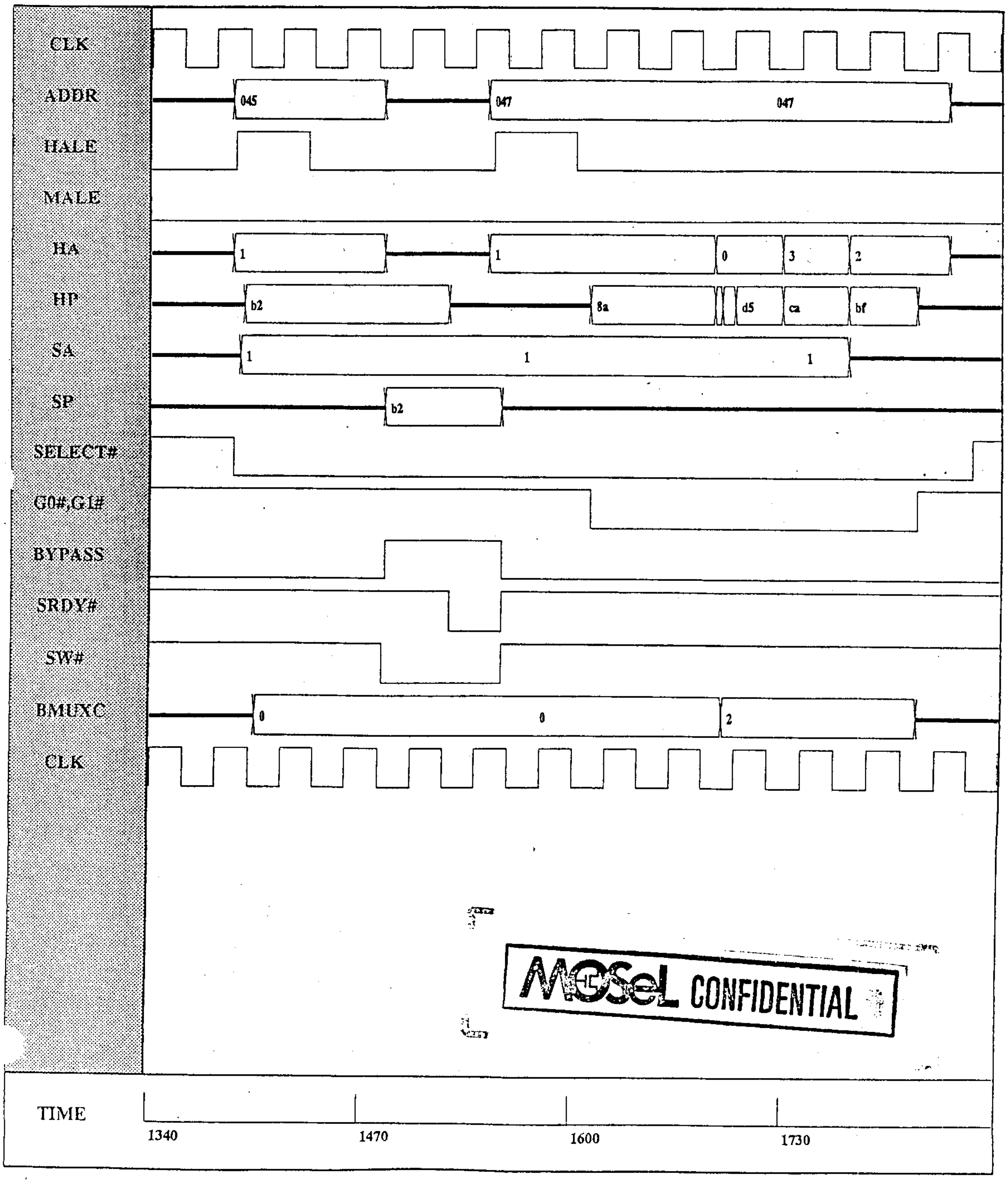


Gateway

Header: Fig.8 Write through write hit with burst read

User: Jason Lin

Date: Feb 16, 1990 14:35:56 Time Scale From: 1340 To: 1860 Page: 1 of 1

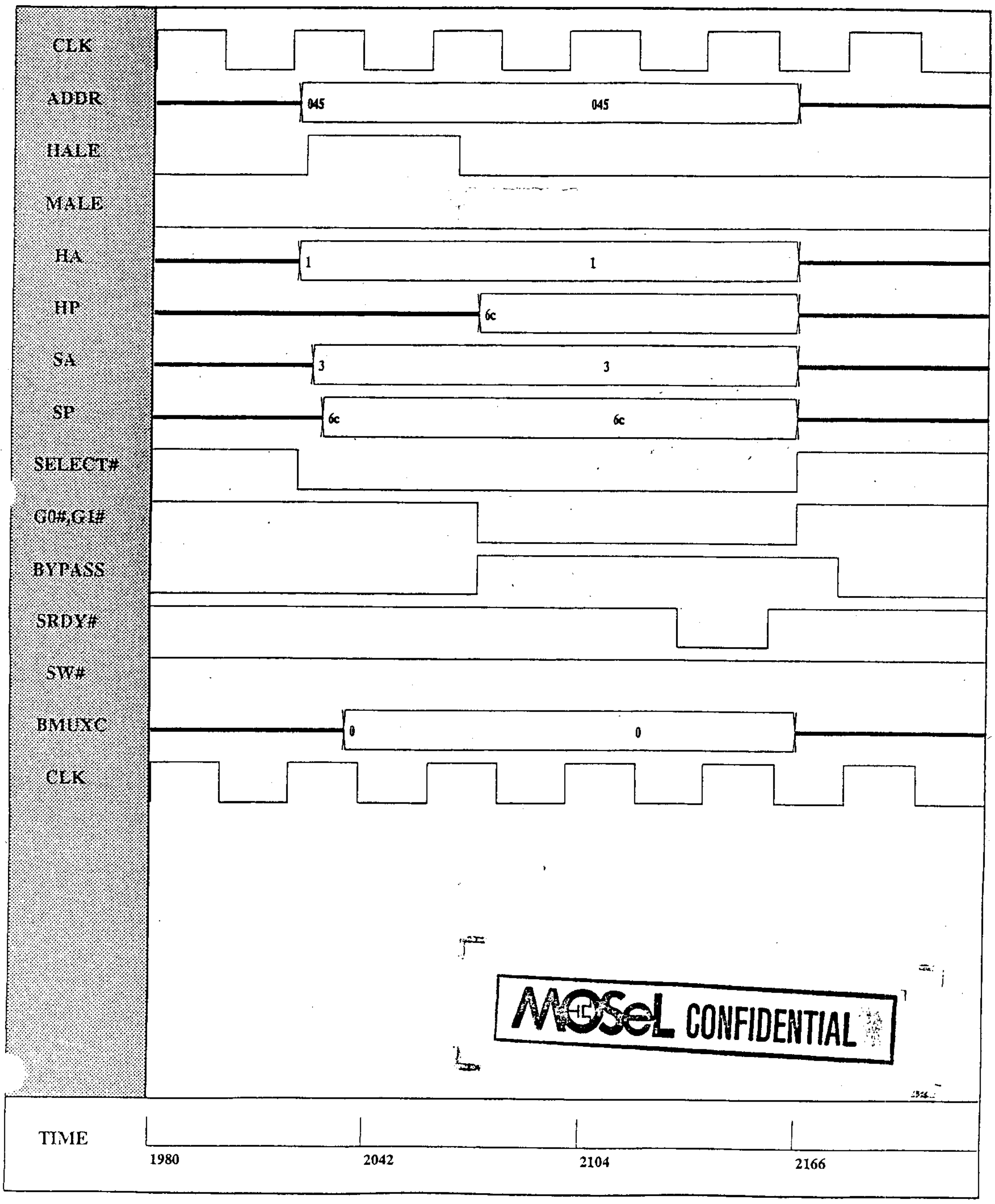


Gateway

Header: Fig.9 Bypass read cycle

User: Jason Lin

Date: Feb 16, 1990 14:26:21 Time Scale From: 1980 To: 2220 Page: 1 of 1

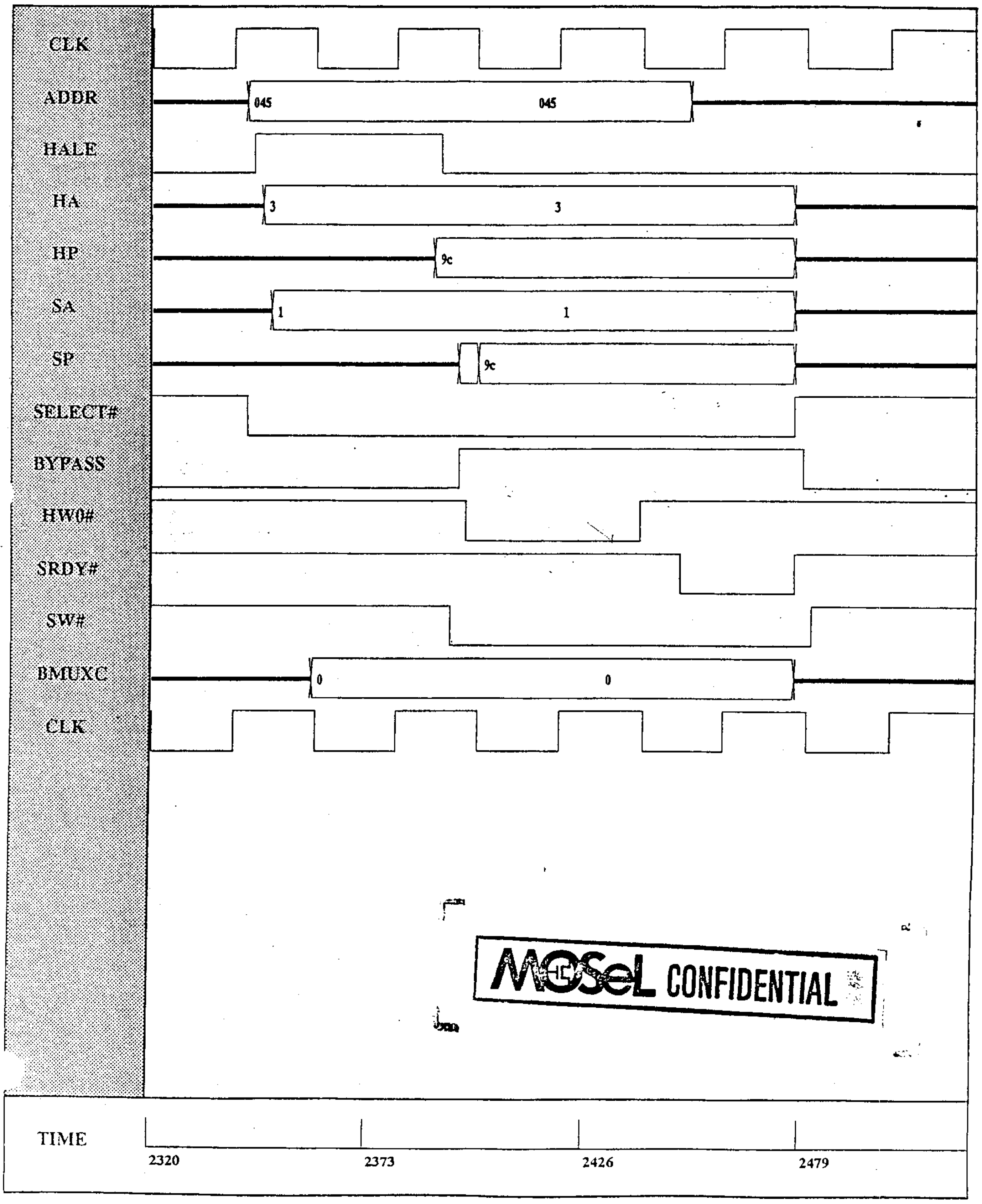


Gateway

Header: Fig.10 Bypass write cycle

User: Jason Lin

Date: Feb 16, 1990 16:20:25 Time Scale From: 2320 To: 2520 Page: 1 of 1



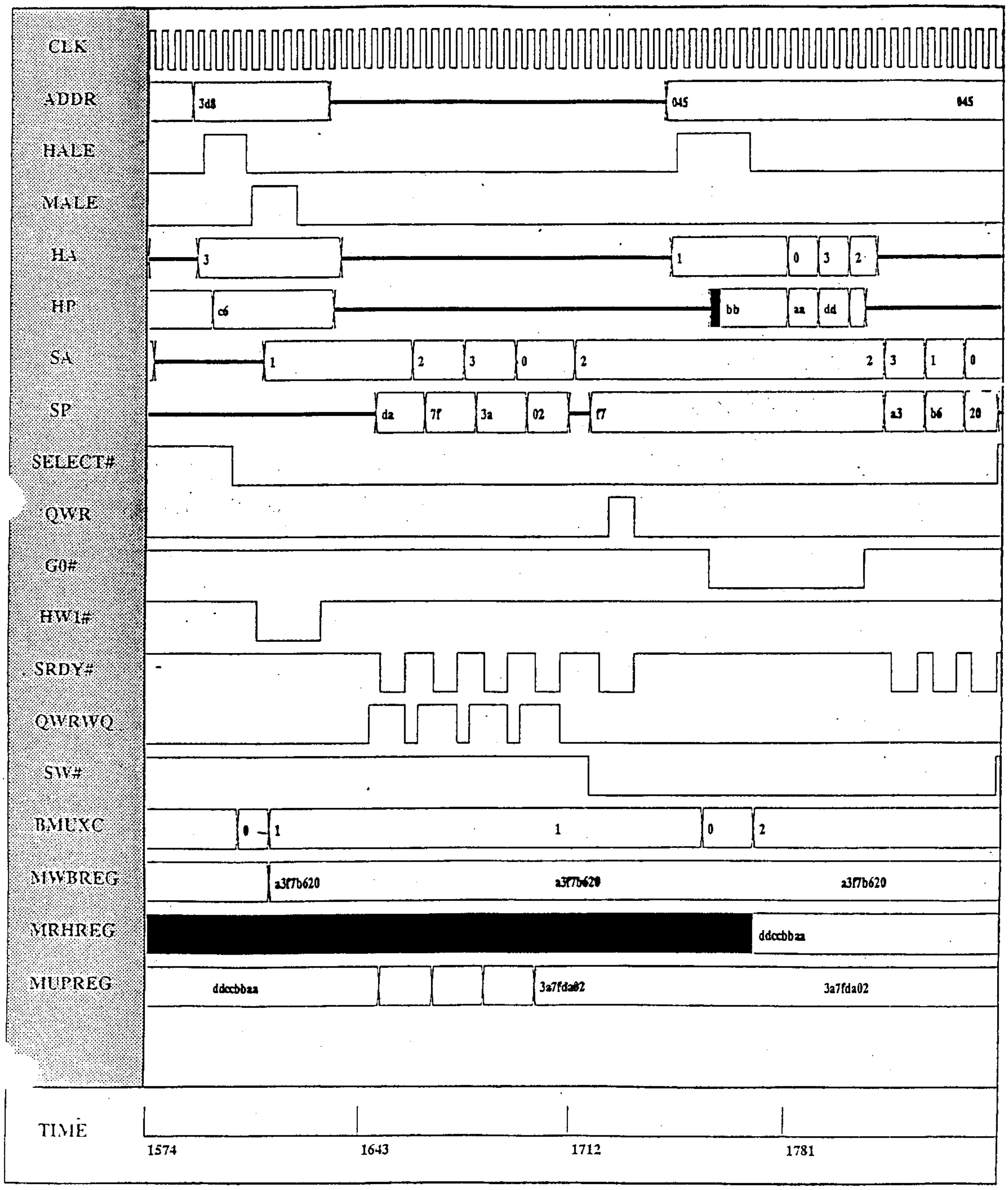
APPEND (PART 3) DOCKET M-1018 US

Gateway

Header: write_miss_with_burst_read

User: Jason Lin

Date: Jan 10, 1990 09:36:26 Time Scale From: 1574 To: 1850 Page: 1 of 1



Gateway

Header: read_miss_with_same_burst_order

User: Jason Lin

Date: Jan 9, 1990 18:25:06 Time Scale From: 1350 To: 1650 Page: 1 of 1

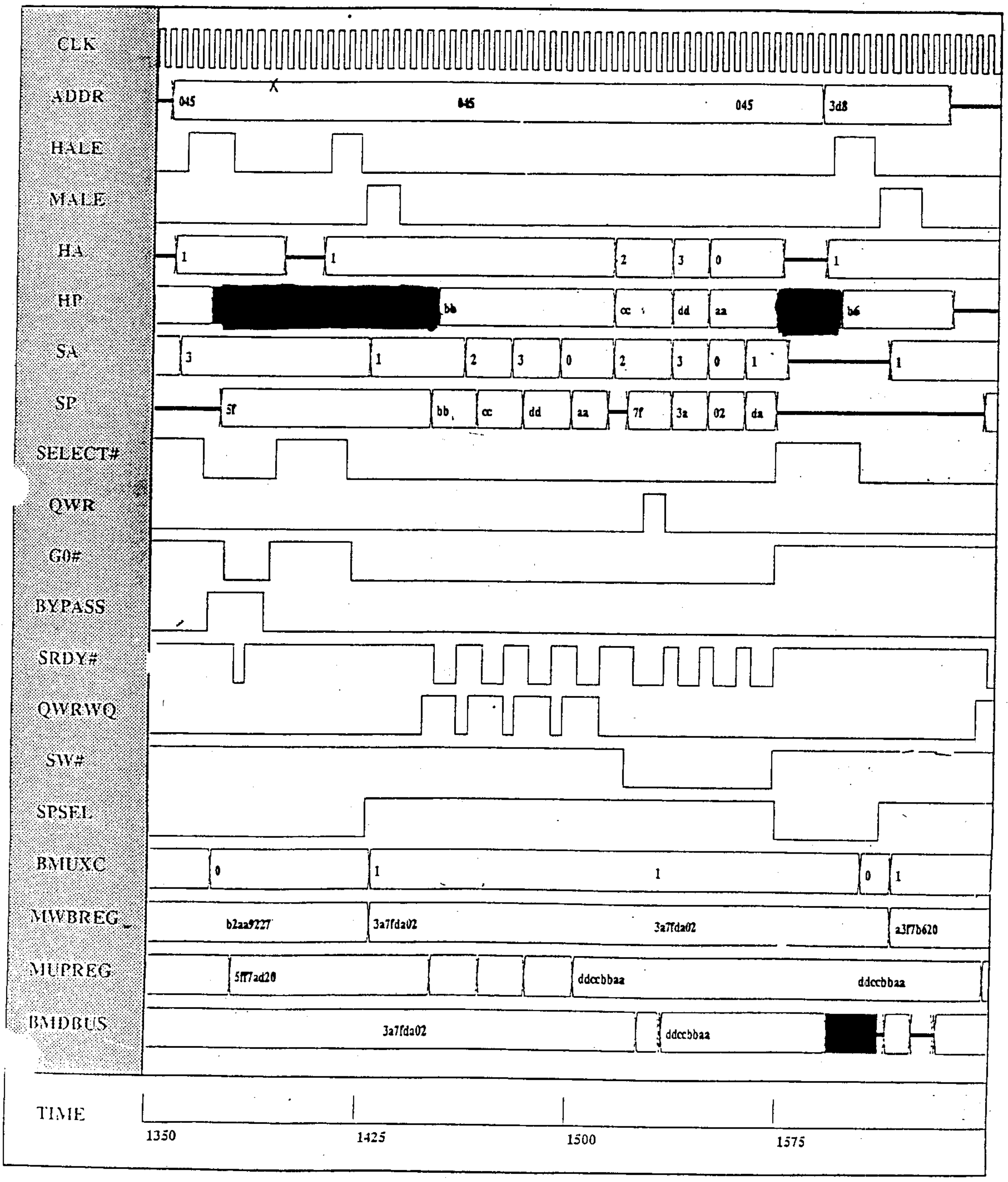


EXHIBIT 5

MOSEL

APPENDIX (part 1)

ADVANCE INFORMATION

MS82C343
Quad DataRam

- o Unique Dual Port Architecture
  - Decouples host and system data buses
  - Cache hits and memory fetches run in parallel to boost hit rate
  - Write-back cycles hidden from processor
  - Processor and system buses can run at different speeds
- o Burst capability
  - System port supports burst or scalar read/write
- o Supports write-back update strategy
- o Supports "demand word first" wrapped around quad fetch order
- o Synchronous 80386 interface
- o High speed access supports 25/33 MHz 80386 systems
- o 2 x 2K x 36 organization
  - Two banks of 2K x 36 divided into four subarrays of 2K x 9
  - Four devices provide 128-bit burst
- o Data ports support 8 bits plus parity
- o On-chip hit and miss address registers
- o System port supports 486 or sequential burst order
- o Supports line abort for decreased miss penalty
- o Word qualify signal prevents same line "dirty" word overwrite by old data

The Mosel 82C343 Quad DataRam is a high-performance CMOS static RAM with a proprietary dual-port architecture optimized for use as a cache subsystem data buffer in 80386 systems. The 82C343 supports quad word data fetches using scalar read and write operations between cache and the host and burst mode reads and writes between cache and main memory to allow the 80386 to run at its full potential. The dual-port architecture allows cache hits to proceed in parallel with main memory accesses, improving hit rate and processor performance. The architecture also permits write-back line replacement cycles to be hidden from the processor, providing an order of magnitude performance improvement over alternative cache implementations.

82C343 BLOCK DIAGRAM

1.0 INTRODUCTION

Cache memory subsystems first saw use in microprocessor-based systems as performance boosters for moderately-fast systems running at clock speeds of 12 to 16 MHz. At these speeds, a microprocessor could still operate out of main memory with reasonable performance. The expense of adding a cache subsystem was only justified by the greatly increased system performance needed for some applications. As microprocessor speeds increased, cache subsystems moved from their status as a luxury to their present status as a necessity. At speeds of 25 to 33 MHz, a processor working out of slower, less-expensive DRAM-based main memory would sacrifice most of its performance to wait states.

In an effort to maximize their performance potential, the newest generation of microprocessors, such as the 80486, are turning to burst memory access to ensure that no delays are encountered in making data available for processing. Such concerns are valid now at 25 and 33 MHz operating speeds, and will become critical as microprocessors move to 40 MHz and above. Cache memories that can handle data bursts from main memory are fast becoming mandatory. In the near future, caches will be required to burst data to the host microprocessor as well.

However, burst capability is not the only factor that affects cache performance. The discussion in Section 3.0 will cover many of the design choices that must be considered when designing a cache subsystem for the current and next generations of high-speed microprocessor systems. The information in Section 2 is a general overview of cache operations and terminology.

2.0 CACHE OPERATION

Cache subsystems store duplicates of recently used main memory information (code or data) in a small, fast, local memory. When cache can deliver a processor memory reference without recourse to main memory, microprocessor operating speed increases and system bus traffic decreases.

The primary measurement of cache efficiency is its hit rate, that is, how often the cache contains the data required by the processor. Cache hits deliver information to the processor without incurring wait states. Cache misses must be referred to main memory, increasing the processor's average access time. To function efficiently, cache must be organized such that the code and data needed by the processor are most often in the cache.

A number of factors affect cache hit rate and must be considered when designing a cache subsystem. External factors include the spacial and temporal locality of code and data (compiler mapping), the levels of multitasking, and context switch frequency. Internal cache issues include cache memory size, block size (number of tags), line size, and degree of set associativity. System performance is also affected by a number of cache management decisions such as the handling of cache misses, memory writes, fetch size, cache coherency, non-cacheable memory, demand-word order, back-to-back read misses, and cache flush.

Before entering a discussion of these design issues, some cache-specific terms should be defined.

2.1 CACHE TERMINOLOGY

Below are some of the words used when describing cache systems and what they mean. Not every cache term is covered.

2.1.1 Page

A page is the unit into which the physical address space is divided, and corresponds to the size of a cache bank (See Figure 2-1). The tag portion of a cache address selects one memory page.

2.1.2 Bank

A bank (or frame) is the unit into which a cache is divided, the number depending on its associativity. Direct-mapped caches have only one bank, two-way set associative have two and four-way set associative have four (See Figures 2-1, 2-2, 2-3). Thus a bank is equal to the cache size in a direct-mapped cache, one-half the cache size in a two-way set associative, and one-quarter cache size in a four-way set associative.

2.1.3 Set Associativity

Associativity is the number of banks in cache into which a memory block may be mapped. Direct-mapped or one-way set associative caches have only one bank into which memory pages may map. Two-way set associative caches have two banks and four-way set associative have four banks, so an incoming block is less likely to overwrite a needed, existing block.

2.1.4 Block

The block is the unit into which pages and banks are subdivided and is the basic unit of cache addressing. The number of blocks in a bank equals the number of cache directory entries (tags in the tag RAM) divided by the cache's set associativity. For a direct-mapped cache with 2048 directory tags, the single bank has 2048 blocks. If that cache is now organized as two-way set associative, each of two banks would have 1024 blocks. Each block is composed of a number of contiguous lines.

2.1.5 Line

The line (or sub-block or transfer block) is the basic unit of data transferred between the cache and main memory. A "hit" or "miss" decision is based on the presence or absence of a line within the cache. The number of lines in a block and the line size is determined by the number of valid bits in each directory entry (See Figure 2-1) or by cache controller convention (for example, a directory entry may have enough valid bits to give each 32-bit doubleword in the block a valid bit, but by convention the controller operates on four doublewords as a line).

2.1.6 Tag

A tag is the part of a directory entry that contains the memory page address from which that particular block was copied. Any block with the same offset within a page may be mapped to the same offset location in a bank. The tag identifies which page the block came from (See Figure 2-1).

2.1.7 Set

A set is all of the directory entries associated with a particular block offset. The number of sets equals the number of blocks. In a two-way set associative organization, a set has two entries, each pointing to a different bank. In a four-way set associative organization, there are four entries in each set.

2.1.8 Line Valid Bit

Each line in a block is represented in it's block's directory entry by a line valid bit. Line valid bits are set when the line is written and are cleared when the block tag changes or when a snoop hit indicates that the line has been changed in main memory by another device (DMA, another processor, etc).

2.1.9 Memory Update

The most widely used methods of updating main memory after a cache write hit are write-through and write-back. In write-through, main memory is automatically updated at the same time the cache is written. The processor must wait until the write is completed before it may resume execution. A variation, usually called "posted" write-through, uses a buffer into which the write data is latched while the processor continues execution. The latched data is then written to main memory whenever the system bus is available.

In a write-back cache, new data written to cache is not passed on to main memory until it is about to be overwritten by a main memory fetch. This greatly reduces bus traffic, but is more complicated and costly to implement.

3.0 CACHE MANAGEMENT POLICIES

There is no single, clear-cut, best way to design a cache. For each application, cost/performance tradeoff decisions must be made concerning design implementations and management policies. In addition, most of these areas are interdependent, so decisions cannot be made in isolation.

Choosing cache size is usually the first decision, but set associativity must be considered at the same time. In general, hit rates improve as cache size increases. On the other hand, large cache arrays consume more power, more board real estate, and more money. Set associativity, however, will improve the hit rate of a smaller cache. So, while having too small a cache will definitely hurt the hit rate, a moderately-sized cache using two- or four-way set associativity can equal the performance of a larger direct-mapped cache. That is because blocks from different pages of memory with the same page offset map into the same space in a direct-mapped cache, while a set associative cache has multiple banks into which a block can be mapped. Therefore a given block is more likely to be present in a set associative cache and is less likely to be overwritten.

Overwriting can be a big problem with direct-mapped caches, and is called "thrashing". Because only one block with the same offset can be in cache at any time, if a program jumps back and forth (thrashes) between blocks with the same offset but different tags, the hit rate approaches zero. Direct-mapped caches are simpler and therefore cheaper to implement, but the one block limitation can exact a performance penalty.

Larger caches are also desirable in a multitasking environment, where different tasks must share the same cache. But even here, four-way set associativity will lessen the chance of one task overwriting the important data of another task. In general, a smaller cache using set associativity will have equal or better performance than a larger, direct-mapped cache.

Block size is also a consideration. A smaller block requires fewer bus cycles to fill and if it's replaced, a smaller amount of old data is overwritten. But the block size is dependent on the number of tag entries in the controller directory and as cache size increases, block size will increase unless the number of tag entries increases. Most controllers have a fixed number of tags, so the block size increases as the cache size increases.

Line size is usually some multiple of the host data bus size, and may equal the block size. The larger the line size, the higher the hit rate. However, system performance may suffer due to the increased number of bus cycles and wait states that a larger line size may require. This disadvantage could be offset by moving data in bursts or by hiding the cache fill cycles from the processor. Line size is also affected by total cache size if the number of tag entries remains fixed. This is because the number of line valid bits also remains the same, so as the block size increases, the line size must do so as well.

System bus traffic is also a primary consideration in choosing a memory update scheme. In a write-through cache, main memory is automatically updated on any processor write. This keeps main memory current no matter how many bus masters are in the system and is an easy policy to implement. On the negative side, the processor must wait until the main memory access is complete before it can continue execution, involving at least as many wait states as a read miss and many more if there is heavy bus traffic. This is particularly true in a multiprocessor system where all caches are using write-through. The total number of writes generated may overload the system bus, creating long processor waits.

With the "posted" write-through variation, the processor usually sees no wait states, but independent logic is needed to complete writing the latched data. If another write should occur before completion of the previous write, the processor must wait until the latch is clear before resuming execution. In addition, because write-through updates main memory on all writes, even temporary results are written out, creating more bus traffic than necessary.

Write-back caches avoid this problem by not updating main memory until new cache data is about to be overwritten by a main memory fetch. This greatly reduces bus traffic, but is more complicated and costly to implement. The controller must keep track of which locations in cache have data in them that has not been written to main memory. This is done by maintaining "dirty" bits for lines and blocks. When a block or line needs to be overwritten by a main memory fetch (because of a cache miss), it's dirty bit is checked. Dirty data must be written to main memory before the fetch can be completed. If an entire block must be replaced, the processor will be held up for many cycles. Write-back also has a more difficult time in a multiprocessor system when another processor requests information that has been updated in a local cache but not written to main memory. To maintain coherency (ensuring that stale data is not accessed), the system must track such requests and must be able to refer the request to the proper location in the proper cache instead of accessing main memory.

Coherency is usually maintained by either software or hardware means. Software-maintained coherency is handled by the operating system software and can be done in a number of ways, but all use host processor time and resources to do the monitoring. It also ties the cache to a particular operating system.

More widely used is the hardware scheme called bus watching or bus snooping. In this case the system address lines are monitored for memory writes and if a write address matches a cached location, that location's valid bit is cleared. As noted above, the "snoop" bus can be used to monitor main memory reads in multiprocessor systems so that "dirty" cache data is sent instead of stale main memory data. The snoop bus can also be used to maintain coherency back to a microprocessor's internal cache, as, for instance, the 80486's "cache invalidation" cycles.

Non-cacheable memory accesses can also be handled by either hardware or software means. The hardware approach usually involves adding address decoders for memory areas that will be non-cacheable. The output of these decoders is sent to a pin on the cache controller which then handles that access as non-cacheable. This approach allows the decoding of any number of memory regions of whatever size, but requires more devices, board space, and power, and the incoming signal from the decoders must be properly timed into the access.

Another approach is to program the non-cacheable regions into registers in the cache controller. This eliminates the need for external decoders but the number of regions is restricted by the number of available registers and their granularity might also be restricted. Other benefits, such as declaring some regions as read-only, are also possible with the software approach.

4.0 PRODUCT OVERVIEW

The 82C343 Quad DataRam is part of the 82C340 chip set which is designed to fully support the 80386 microprocessor with a write-back cache. The 82C343 forms the highly integrated backbone of an innovative high-speed data path which offers significantly higher performance. Its proprietary dual-port architecture permits processor and main memory accesses to occur in parallel, while hiding write-back cycles from the processor, contributing to a substantial performance increase over alternative implementations.

The 82C343 also supports quad word data fetch and, though not common in 386 systems, burst operation with main memory. In an 80386 system which is designed for future upgrade to an 80486, the main memory subsystem may use Intel's strongly suggested 64-bit, bank interleaved main memory organization. This memory organization is accessed using the non-sequential burst order used by the 486. The 82C343 directly supports that burst order to and from main memory, as well as the sequential burst order used with the smaller, less expensive, and standardized 32-bit sequential memory organization. In either case, the 82C343 supports "demand word first" wrapped around burst or quad fetch order, so the processor is never delayed while waiting for the demand word.

4.1 ARCHITECTURAL OVERVIEW

The 82C343's highly integrated dual-port architecture simplifies the design of any cache system by incorporating most of the functions that have previously required external components, including address latches and the data transceiver. Its dual 9-bit ports allow the use of parity between the processor and cache and main memory and cache. When four Quad DataRams are used in a normal 32-bit configuration, they support a 64K byte two-way set associative or direct-mapped cache capable of a 16 byte burst to either main memory (See Figure 4-1). The Quad DataRams are connected up such that the first device supplies the first byte of each of the four doublewords in a line. Likewise, the second device supplies the second byte, the third supplies the third byte, and the fourth supplies the fourth byte for each doubleword. The set of four subarrays labeled A will contain the 32-bit doubleword located at the line's highest address. Subarrays B, C, and D contain the third, second, and first doublewords in the line, respectively.

4.1.1 Array Organization

The 82C343 is arranged internally as two banks of 2K X 36 bits which are in turn subdivided into four subarrays of 2K X 9 (See Figure 4-2). The two bank division fits well with either two-way set associative or direct-mapped cache organizations. More Quad DataRams can be added for larger caches or four-way set associativity. When used in sets of four, internal data transfers move four doublewords (128 bits, on reads or writes, achieving a true 128-bit quad word transfer in only one clock. When doing memory burst operations, the 82C343's unique internal structure behaves like a static column RAM in that the input buffer and word line delays occur only on the first access to the first subarray. Subsequent accesses in the burst are supplied by successive subarrays which are pre-addressed. This structure allows full speed burst accesses without the use of 10 nsec SRAMs.

4.1.2 Address Paths

The 82C343's address path is also unique. It contains two on-chip address latches, one for hit addresses and one for miss addresses. The hit address latch contains the address information for the initial access. If, on the next clock the access is found to be a miss, the address information is copied into the miss address latch. The hit latch is now available immediately for subsequent cache accesses so the processor may continue execution immediately. Meanwhile, the miss latch provides addressing to the internal array for storage of the incoming fetch data when it arrives.

Another part of the 82C343's address path is made up of the word-enable address pins for the host (HA<3:2>) and system (SA<3:2>) ports. These pins are driven by the cache controller and are used to select the word address within a quad word. For the system port, the SA pins set the burst order for main memory accesses

4.1.3 Data Paths

The 82C343's proprietary dual-port architecture is the key to its exceptional performance. By decoupling the host and system data buses, processor cache accesses and main memory accesses can proceed simultaneously. The effects of this decoupling on a write-back cache are dramatic. It means that write misses are handled with zero wait states because read and write cache hits can proceed in parallel with write miss data fetching. Moreover, when combined with the Quad DataRam's internal mechanism for saving write-back data during read or write misses, the dual-port architecture allows write-back line replacement cycles to be completely hidden from the processor.

This is because the Quad DataRam's innovative architecture eliminates the historical disadvantage of write-back caches: having to do a replace cycle before doing the data fetch. With the 82C343, the replace data is saved as soon as a miss is detected and the fetch is begun immediately. Only after the fetch is complete does a replace cycle begin, and during that time, the processor can resume accessing the cache array through the host port. As an example, in a system with two wait state memory and non-burst fetches, the use of Quad DataRams would allow the 386 to be operating at full speed after a read miss in just 16 clocks. Using standard SRAMs, the same system would take 48 clocks (32 clocks for two lines on a tag miss plus 16 clocks for the new data).

Decoupled buses also means the main memory subsystem design is not subject to redesign every time the processor module is upgraded. In fact, the processor data bus and the system data bus can run at different speeds. Since the processor will work out of its cache subsystem more that 95% of the time, main memory need not be redesigned to run at the same speed as the processor. That's especially true if the system uses a burst memory controller so that the Quad DataRams can support burst operation to and from main memory. The main memory system might then work at a reasonably easy to design speed of 20 MHz, while the processor module (processor, cache controller, and Quad DataRams) ran at 25, 33, or even 40 MHz. Even upgrading the processor from an 80386 to an 80486 would involve only minor design changes to main memory.

Both of the 82C343's 9-bit (eight bits plus parity) data ports support "demand word first" wrapped around quad word operations as well as scalar reads and writes. The system port supports burst operations to and from main memory if the system memory controller also supports burst.

If system memory uses the design-intensive 64-bit, bank interleaved architecture recommended by Intel instead of the standard, 32-bit sequential architecture, then the system port would use the 486 burst sequence to main memory instead of sequential order. The 82C343 supports both sequential and 486 burst ordering. In either case, the fetch data from main memory is brought in "demand word first" and that first doubleword passes directly to the 386 through the Quad DataRam's special bypass path. The 386 can then resume execution while the remainder of the burst data is brought in and stored in the array.

5.0 FUNCTIONAL OVERVIEW

The 82C343 fully supports an 80386 using a write-back cache, including support for quad word data fetch, non-cacheable accesses, and multiple replacement cycles for lines longer than 16 bytes. If the cache and system memory controllers support burst operation, the 82C343 will also support burst reads and writes to main memory.

5.1 Read Operations

The 82C343 supports scalar reads, appearing to the 386 to be a standard SRAM interface. If the cache controller can implement quad fetch on a read miss, the Quad DataRam supports a "demand word first" wrapped around fetch order, so the processor is never delayed while waiting for the demand word.

5.1.1 Read Hit

On a read hit, the 80386 address is latched into the on-chip hit address register while the data is being accessed in the array and fed to the host port. A read hit takes a standard two bus cycles to complete.

5.1.2 Read Miss

On a read miss, the address is latched into the hit address register and then copied to the miss address register after miss detection. At the same time, the write-back line being replaced is saved. Because Quad DataRams are being used instead of standard SRAMs, the replacement cycle is delayed and the miss data fetch is done immediately. This reduces both first word fetch latency and total fetch latency. The missed line is fetched from main memory in "wrap-around" fashion with the demand doubleword fetched first and is simultaneously written into the array and sent on to the 386 through the host port using the 82C343's internal bypass path. The fetch may be either burst or scalar, depending on the system's memory management controller. Once the data reaches the 386, the processor may resume execution out of the cache array using the host port while the remaining three doublewords are fetched.

If the line being replaced is the same as the miss line (the demand word was invalid), an external word qualify signal can be provided by the cache controller. As the data comes in on the system port, the word qualify signal prevents any remaining words in the array line which may be valid and dirty from being overwritten by stale data from main memory as the miss line is written into the array.

Once the fetch is complete, if the saved replace line contains any valid and dirty doublewords (whether it's the same line or not) the entire line is sent back to main memory in either burst or scalar mode. The Quad DataRam's dual-port structure allows this operation to be completely hidden from the processor.

5.2 Write Operations

5.2.1 Write Hit

A scalar write hit in write-back mode using Quad DataRams is a very simple operation. The address is latched into the hit address register and the data is latched into the on-chip write register (no external transceiver required). The controller sends back a RDY# signal immediately, releasing the processor in just two bus cycles. The data is then written into the array and the controller marks that location as dirty.

5.2.2 Write Miss

In a write-back system, a write miss is handled much the same as a read miss, except that the processor is not halted while the fetch to main memory is completed. The write data is latched into the write register while the address is latched first into the hit address register and then into the miss address register. The line to be replaced is saved and then the write data is written into the data array. The cache controller marks the location as dirty and at the end of the second bus cycle sends a RDY# to the processor which then continues execution, using the host port to access the cache.

The missed line is now fetched (in either burst or scalar mode) from main memory in "wrap-around" fashion with the stale demand doubleword fetched first. The write data has already been written to the array, so when the stale demand word comes it is masked to prevent overwriting the new data. Likewise, if the line being replaced is the same as the miss line, an external word qualify signal can be provided by the cache controller to prevent any remaining words in the line which may be valid and dirty from being overwritten by stale data from main memory as the miss line is written into the array.

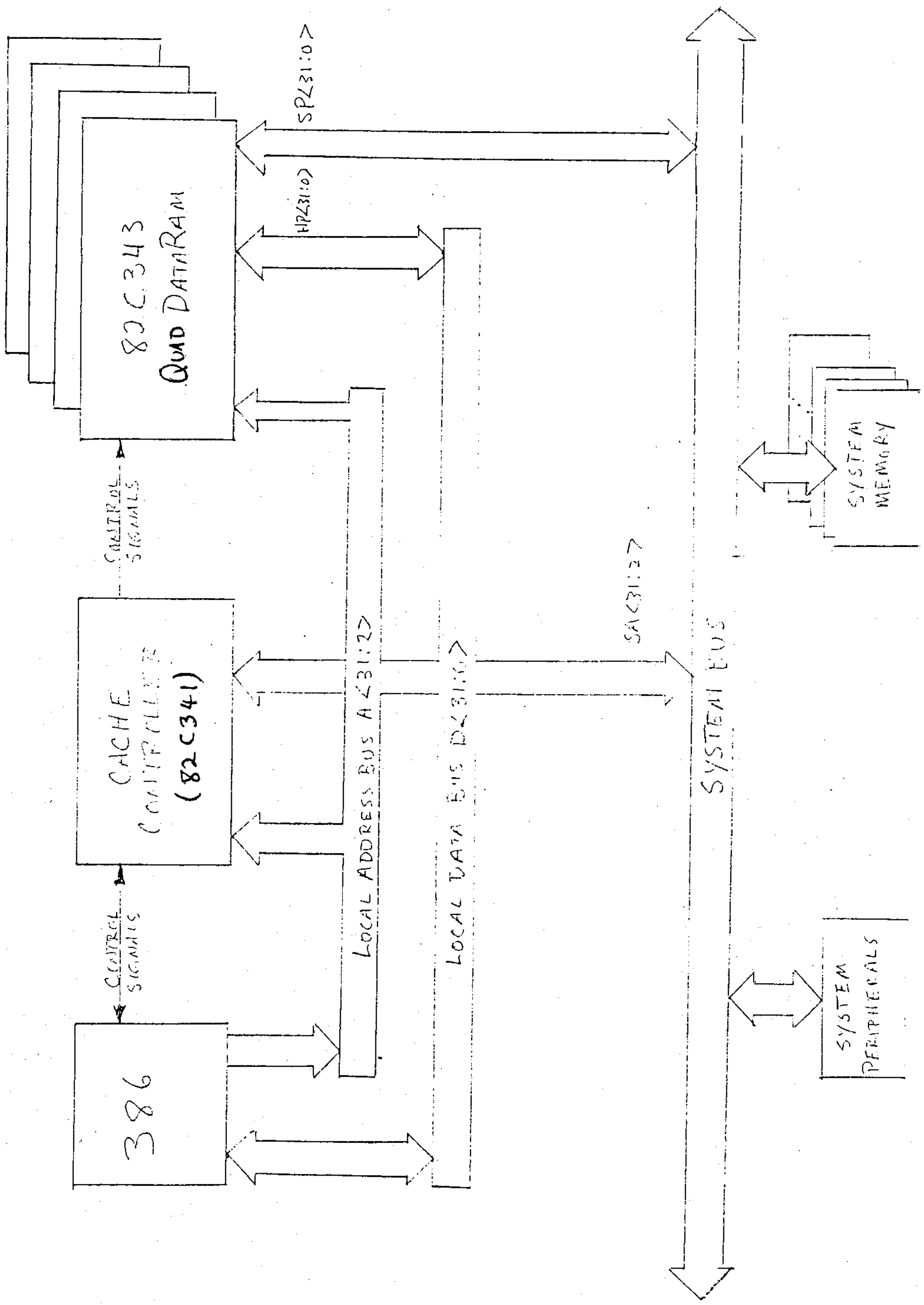
If the saved replace line contains any valid and dirty doublewords (whether it's the same line or not) the entire line is sent back to main memory in a write-back sequence.

5.3 Non-cacheable Cycles

Non-cacheable scalar reads and writes are simply passed through. Writes are handled as write misses and all reads are treated as read misses. In no case is the data array written.

5.4 Multiple Replacement Cycles

Mosel's Quad DataRam will also support cache systems with line sizes larger that four doublewords. During a replacement cycle, activation of an external signal by cache control logic causes an internal counter to access neighboring lines of data so that lines of eight or 16 doublewords can be saved for burst write-back transparently to system memory. This activity does not prevent the read miss data from being fetched first, eliminating what would be a large replacement time penalty.


82C343
QUAD DATARAM
SP<31:0>
HP<31:0>
CONTROL SIGNALS
CACHE CONTROLLER (82C341)
CONTROL SIGNALS
386
LOCAL ADDRESS BUS A<31:2>
LOCAL DATA BUS D<31:0>
SA<31:2>
SYSTEM BUS
SYSTEM MEMORY
SYSTEM PERIPHERALS

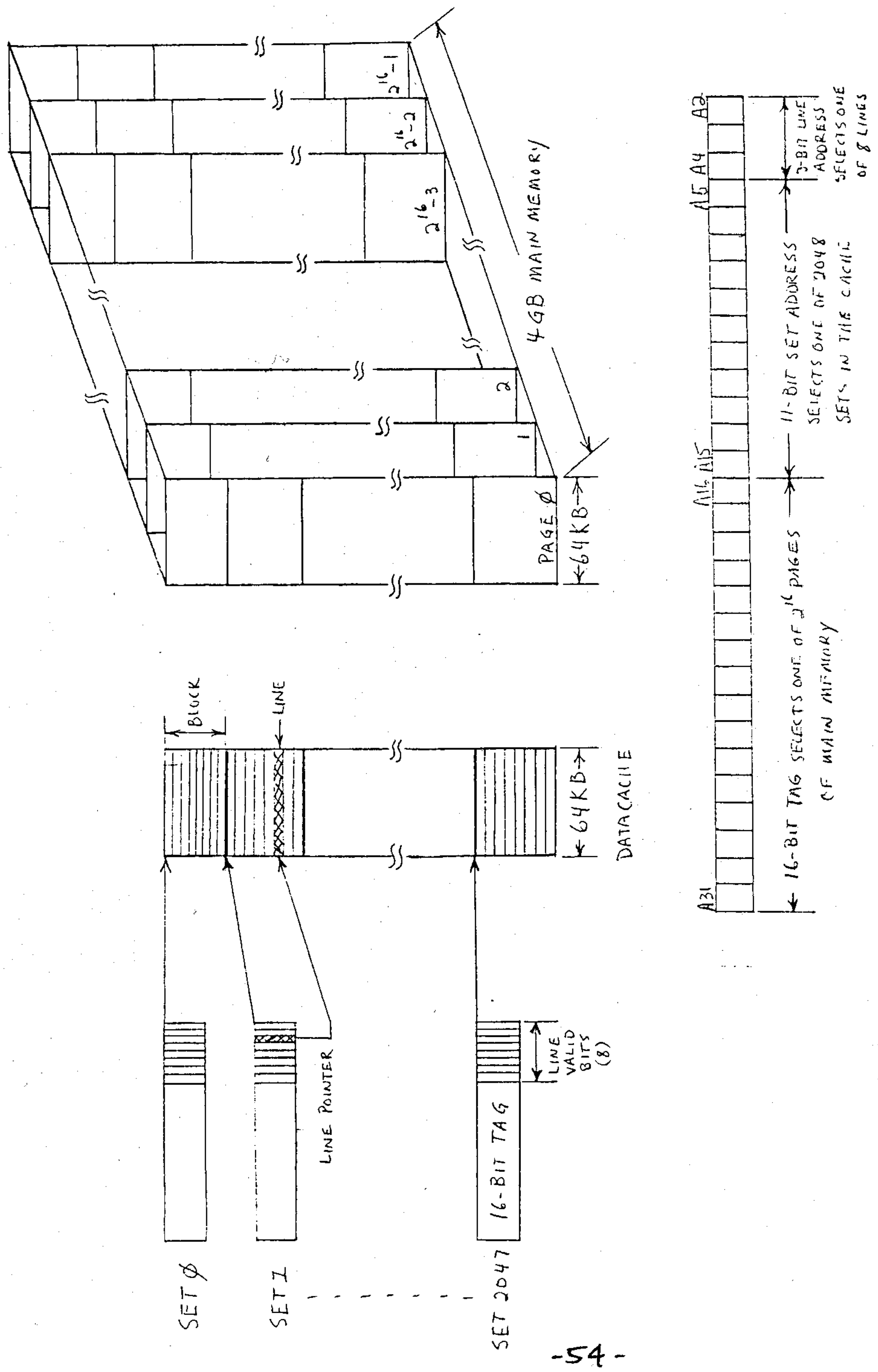


FIGURE 2-1 DIRECT MAPPED CACHE ORGANIZATION AND ADDRESS BUS BIT FIELDS

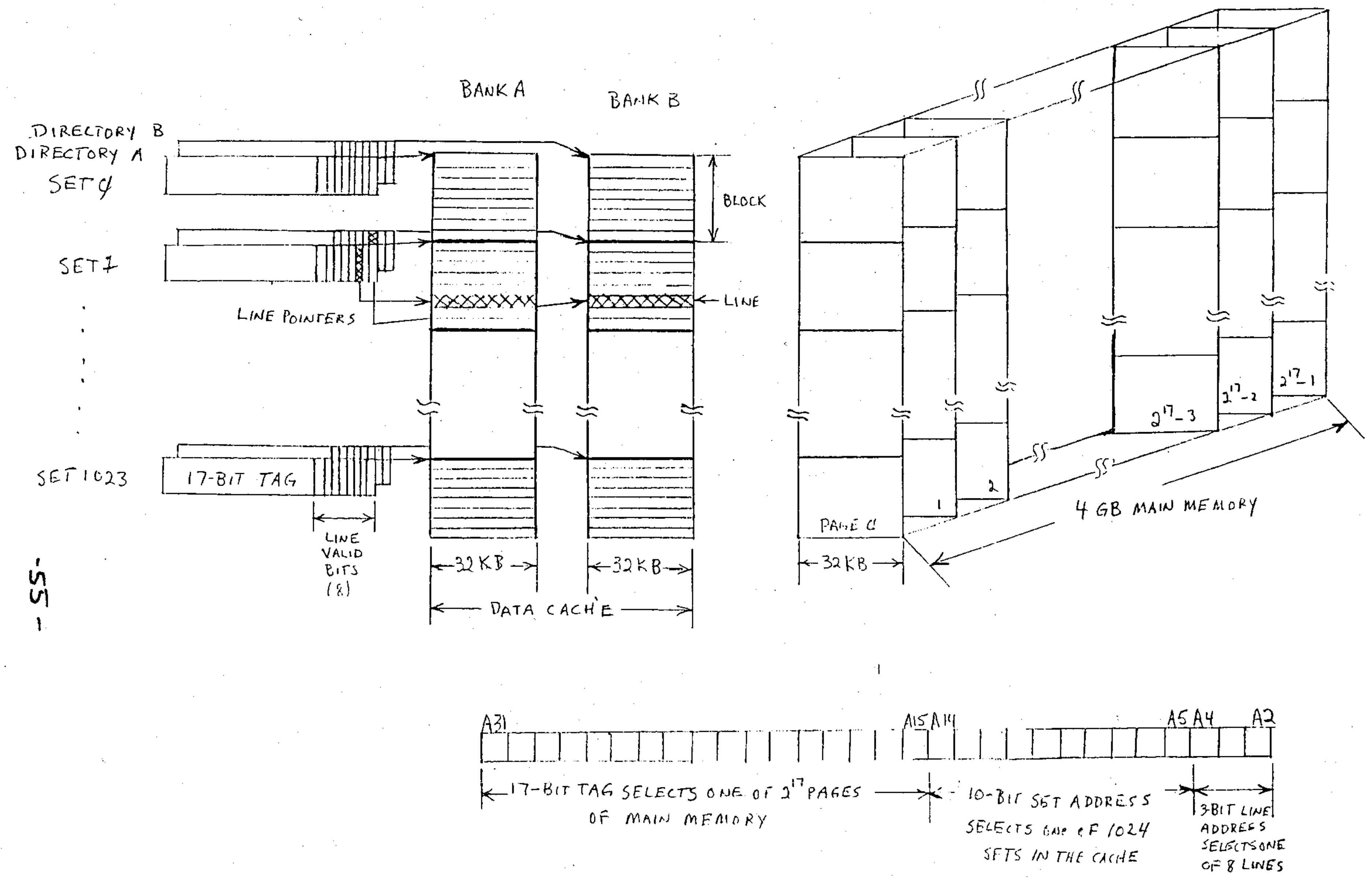


FIGURE 2-2 TWO-WAY SET ASSOCIATIVE CACHE ORGANIZATION AND ADDRESS BUS BIT FIELDS

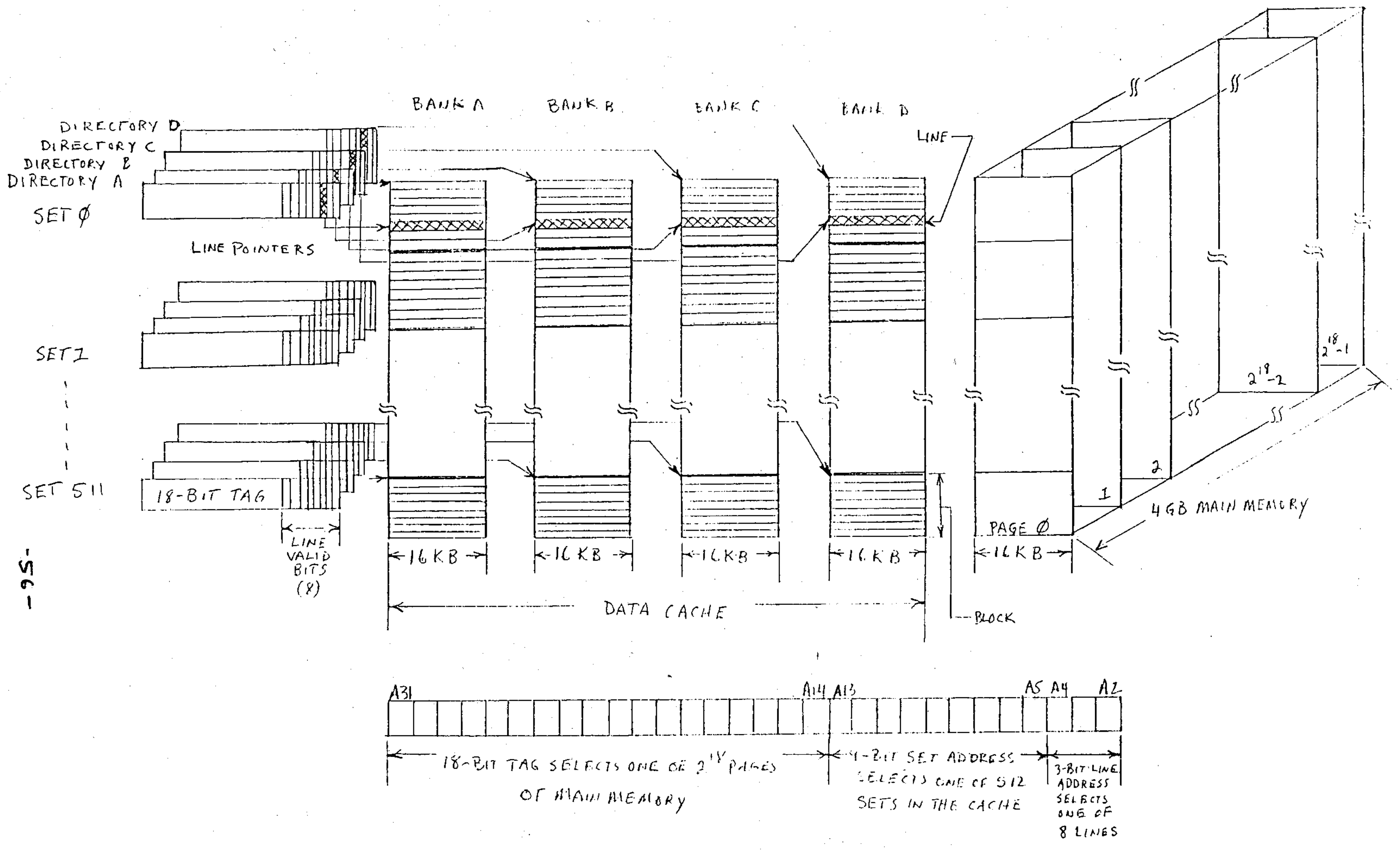
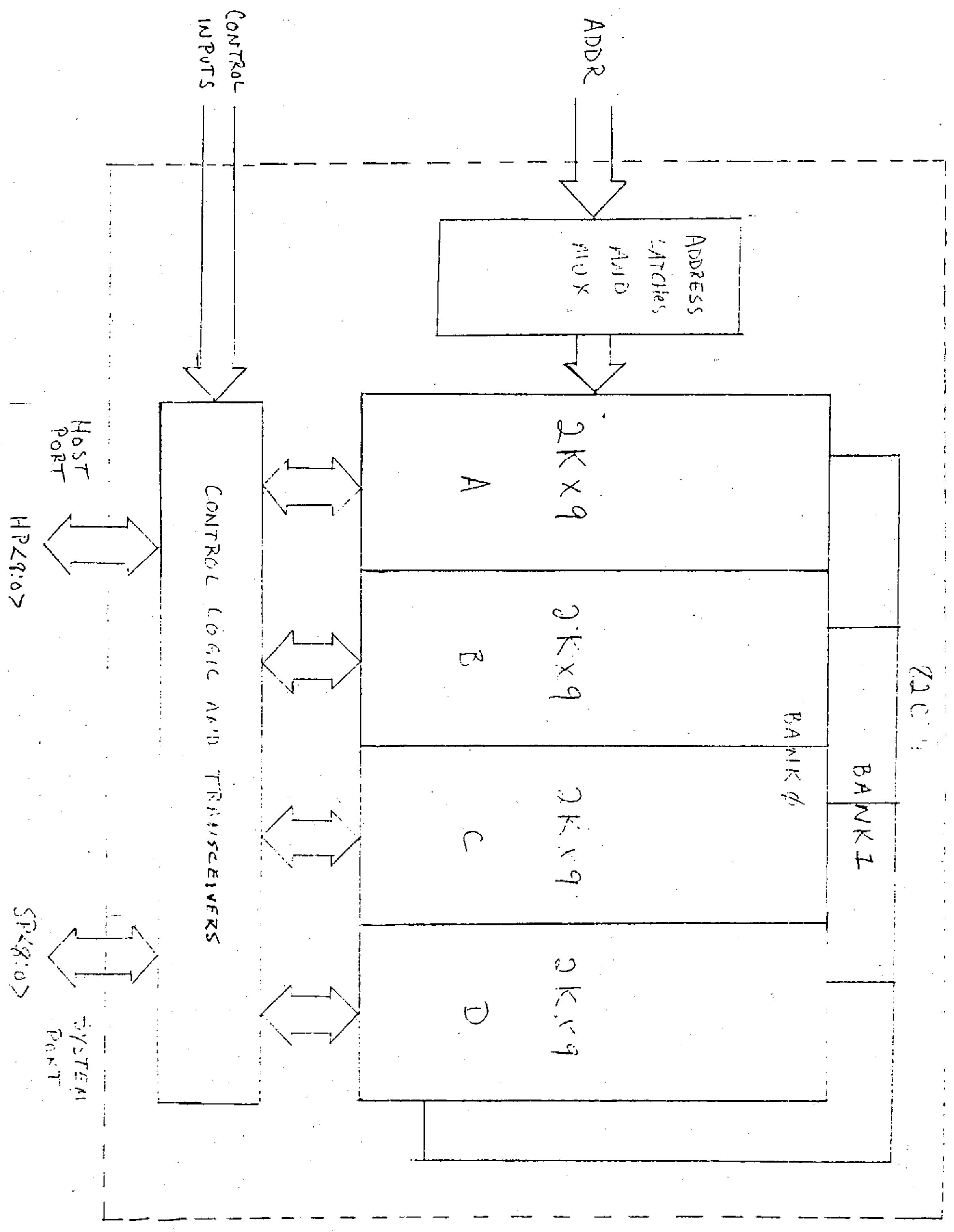


FIGURE 2-3 FOUR-WAY SET ASSOCIATIVE CACHE ORGANIZATION AND ADDRESS BUS BIT FIELDS


ADDR
CONTROL INPUTS
ADDRESS LATCHES AND MUX
2Kx9 A
2Kx9 B
2Kx9 C
2Kx9 D
BANK 0
BANK 1
22C
CONTROL LOGIC AND TRANSCEIVERS
HOST PORT
HP<9:0>
SP<9:0>
SYSTEM PORT

CERTIFICATE OF SERVICE

I hereby certify that on November 6, 2007, I caused the foregoing to be electronically filed with the Clerk of the Court using CM/ECF which will send electronic notification of such filing to the following:

John G. Day, Esquire
Steven J. Balick, Esquire
ASHBY & GEDDES

Additionally, I hereby certify that true and correct copies of the foregoing were caused to be served on November 6, 2007 upon the following individuals in the manner indicated:

BY E-MAIL

John G. Day, Esquire
Steven J. Balick, Esquire
ASHBY & GEDDES
500 Delaware Avenue, 8th Floor
Wilmington, DE 19899

jday@ashby-geddes.com
sbalick@ashby-geddes.com

BY E-MAIL

Sten A. Jensen, Esquire
HOGAN & HARTSON LLP
555 Thirteenth Street, NW
Washington, DC 20004

sajensen@hhlaw.com

BY E-MAIL

Steven J. Routh, Esquire
HOGAN & HARTSON LLP
sjrouth@hhlaw.com

William H. Wright, Esquire
HOGAN & HARTSON LLP
whwright@hhlaw.com

William C. Gooding, Esquire
GOODING & CRITTENDEN, L.L.P.
billgooding@gooding-crittenden.com

/s/ Mary B. Graham

Mary B. Graham (#2256)